

SYSTEM MEMORY PROTECTION AND VULNERABILITY ASSESSMENT IN PRESENCE OF SOFTWARE ATTACKS

By

MINGBO ZHANG

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Saman Zonouz

and approved by

New Brunswick, New Jersey

MAY, 2021

ABSTRACT OF THE DISSERTATION

System Memory Protection and Vulnerability Assessment in Presence of Software Attacks

by Mingbo Zhang

Dissertation Director:

Saman Zonouz

Software vulnerabilities widely exist among various software from operating system kernel to web browser, from PC to embedded device. The arms race is continuing between new vulnerability exploit techniques and new mitigations. The essential part of protecting software from compromising relies on system memory protection in specific ways. Addressing protection of system critical variables, heap layout, and user variables that are referenced freely from the kernel are the state-of-art challenges. This dissertation aims at protecting the above-mentioned vulnerabilities that exist in the wild and presents systematic mitigation solutions. For each specific vulnerability, our mitigation either leverages a new CPU features such as Intel SGX or an existing CPU feature in a novel way to achieve adequate protection with a modest performance overhead. Additionally, we utilize a software-only method to solve the use-after-free vulnerability in the web browsers — a trade-off between the deterministic heap layout and memory usage. Furthermore, we develop a new software attack that is parasitic on an extra piece of hardware circuit to assess conventional software mitigations' effectiveness. We evaluated each system with real-world vulnerabilities and their exploits that are publicly

available. The results show that these mitigations can effectively protect the system with an acceptable performance overhead. Our parasitic-hardware-based attack reveals the possibility of being deployed in the field devices such critical controllers (e.g., programmable logic controllers PLCs) in cyber-physical platforms such as the power grid infrastructures. This type of attack completely evades conventional software mitigation techniques.

Acknowledgements

I would like first to thank my doctoral committee: **Prof. Chen**, **Prof. Ortiz**, **Prof. Wei**, **Prof. Yuan**, and **Dr. Schulte**.

I took courses and advices from you, thank you for teaching me!

Special thank you to my advisor **Prof. Zonouz**. You opened the world to me!

Dedication

To my wife Qin and my two kids Ray and Summer.

Thank you for bearing with my bad temper.

Table of Contents

Abstract	ii
Acknowledgements	iv
Dedication	v
List of Tables	ix
List of Figures	xi
1. Introduction	1
2. Dynamic Memory Protection via Intel SGX-Supported Heap Allocation	5
1. Introduction	5
2. Background and Case Studies	8
2.1. Heap Metadata Vulnerabilities	9
2.2. Non-Control Flow Attacks	11
3. MetaSafe Overview	13
4. Threat Model	14
5. Design	14
5.1. Program Isolation	14
5.2. Secure Heap Management	17
5.3. Protection of User Program Critical Data	20
6. Metasafe Implementation	22
7. Evaluation	24
8. Related Work	30
9. Conclusions	33

3. Use-After-Free Mitigation via Protected Heap Allocation	34
1. Introduction	34
2. ZEUS's Design	37
2.1. Optimization	40
2.2. Security	41
3. Implementation	43
4. Evaluation	44
4.1. Performance	45
4.2. Memory	46
4.3. Security	47
4.4. Case studies	48
4.5. Complementing Isolated Heap	52
5. Conclusion	54
 4. Effective Mitigation for Kernel-Level Time-of-Check-to-Time-of-Use	
Vulnerabilities	55
1. Introduction	55
2. Background	58
2.1. Kernel-level TOCTOU Vulnerability	58
2.2. Supervisor Mode Access Prevention (SMAP)	59
2.3. Intel Virtualization Technology	61
2.4. x86 Architecture	62
3. Overview	66
3.1. Threat Model	66
3.2. High-Level Design and Challenges	67
3.3. Approach Overview	68
4. Finding Kernel TOCTOU Bugs	69
5. SMAPRO Design	72
5.1. System Module	72

5.2.	Hypervisor	78
6.	Implementation	79
7.	Evaluation	83
7.1.	Performance	84
7.2.	Case Study	88
8.	Discussion	90
9.	Related Work	92
10.	Conclusion	93
5.	Targeted Attacks against Cyber-Physical Infrastructures via Distributed	
	Hardware Implants	94
1.	Introduction	94
2.	Background	97
3.	Overview	100
3.1.	Adversary Model	101
3.2.	Challenges and Approaches	102
4.	Reverse Engineering and Design	104
4.1.	Reverse Engineering	104
4.2.	Design	116
5.	Implementation and Experiment	117
6.	Evaluation	120
7.	Related Work	125
8.	Discussions and Mitigations	127
9.	Conclusions	128
6.	Conclusion	130
	Bibliography	132

List of Tables

3.1. ZEUS's overhead on browser's memory (%).	46
3.2. Vulnerabilities used in the evaluation.	47
4.1. Recent vulnerabilities categorized as race condition or time-of-check-to-time-of-use in the CVE database.	59
4.2. System call count and user-pages accessed for GUI & non-GUI programs	86
4.3. In the selected double-fetch results, for every two lines, they have the same CR3 and TEB, which indicate the two records are from the same process, same thread. The two EIP are shortly apart, but the addresses they reference are the same user-mode address.	87
5.1. LM3S2793 Memory Map. Only list the address space of the memory and devices related to this paper. FiRM-compliant (compliant to the ARM Foundation IP for Real-Time Microcontrollers specification).	106
5.2. LM3S2793 ROM API table is at a fixed address 0x1000010, and each table entry is 4 bytes address that points to a second-level table, which corresponds to a class of peripheral devices.	108
5.3. GPIO API Table. Each table entry is the entry address of an API, and the function parameters are passed through registers. There is no privilege or mode change when calling into the APIs.	109
5.4. The internal registers of the PD9593 are selected by the master device sending the command byte. To control the LED, we need to operate the control register and the output register and keep the two polarity inversion registers' default value.	114

5.5.	DPACC is used for Debug Port (DP) accesses. APACC is used for Access Port (AP) accesses, and it can access a register of a debug component of the system to which the interface is connected.	118
5.6.	Debug Port registers. Debug Port only has one bank, a total of four registers, which can be specified by A[3:2] of the DPACC register. . . .	119
5.7.	Part of Memory Access Port (MEM-AP) registers. MEM-AP has 16 banks, and each bank has four registers, which can be specified by A[3:2] of the APACC register. The bank needs to be specified by the DP:SELECT register.	119

List of Figures

2.1. <code>malloc_chunk</code> structure	10
2.2. With only 4 or 8 bytes overflowing , it is enough to overwrite the size field of next free chunk, which will enlarge the next free chunk. The arrow indicates where the metadata corruption happens	11
2.3. After attacker overwriting size field of the adjacent free chunk. When next time calling <code>malloc()</code> , the program get a over large buffer which contains part of other buffer. In GHOST’s case, that part belongs to an internal memory allocator and could be further exploited.	11
2.4. High-Level Overview of METASAFE	13
2.5. The same size region are within one run, regions are next to each other, no metadata between them	19
2.6. <code>sgx_malloc()</code> metadata is stored inside one enclave, in only contains the information that where the run is located in memory and which region in that run is used	20
2.7. Memory layout of user program protection through enclave.	21
2.8. <code>sgx_malloc()</code> overhead in five non-trivial open source application. . .	24
2.9. Local memory access, commonly used IPC methods and SGX’s performance, each access is repeated 10000 times.	25
2.10. METASAFE implementation of OpenSSL with SGX enclave. The RSA private key is stored in the enclave, while METASAFE provides an interface to the internal CDA APIs that facilitates typical functions with the RSA private key, such as decryption. This mechanism would protect the RSA public key in spite of zero-day exploits such as the Heartbleed vulnerability.	27

3.1. Region-based allocations with random prefix.	39
3.2. ROP chain after unpredictable heap spray.	40
3.3. Wasted memory used for random allocation.	40
3.4. Firefox heap allocation request size frequency.	44
3.5. ZEUS's performance overhead with different allocation request sizes. .	45
3.6. ZEUS's performance overhead with loading different websites.	46
3.7. Exploit success probability against ZEUS.	47
3.8. Tor's memory snapshot after the heap sprays that surround the target buffer with crafted data.	49
3.9. Call sequence for UAF attack in Tor browser.	50
3.10. Exact manipulation of several addresses (pointer values) is required for CVE-2016-9079 attack success.	51
3.11. Tor browser's memory with ZEUS enabled.	52
3.12. Exploit success rate for CVE-2016-9079.	53
3.13. Internet Explorer CDOMTextNode object in memory. Object's offset 0x30 (boxed) is attacker-controllable memory address.	53
3.14. ZEUS prevents Microsoft IH UAF exploits.	54
4.1. Pseudocode of the vulnerability fixed in ms08-061. The vulnerable vari- able is in red. The kernel reads it twice, and it may get a different value for the buffer allocation and the subsequent buffer copying. It is com- mon to see such a coding style. However, it is vulnerable because the two reads cross the privilege boundary.	60
4.2. Thread zero invokes the vulnerable system call. The attack windows lie in between the two kernel reads, namely, the two instructions that underscored. The attacker can repeatedly open the attack time window by calling the system call. Simultaneously, thread one flips the high bit of the user-mode variable in a loop. The two threads compete, hoping an odd number of flips occur during the time window. It enlarges the variable; otherwise, the variable remains the same.	61

4.3. Linear-Address Translation to a 4-KBbyte Page using 32-Bit Paging [70]	63
4.4. The hypervisor is capable of confining the system-wide feature SMAP into one process. When the hypervisor catches the process context switch events, it changes the SMAP enable bit in the CR4 register to only set during the target process. The processor raises page fault exception when the kernel accesses userspace so that we can protect those pages. Thread 1 can read a protected page but can not write. The read is allowed by automatically setting the protected page back to userspace with a read-only permit. However, when the write instruction raises an exception, the page fault handler suspends the thread until the system call ends.	69
4.5. CR3 0x6d40320; TEB 0x7ffdd000; EIP 0xbf812de4 and 0xbf812e4b read the same user-mode address 0x4808c4 within one system call.	71
4.6. Page attributes transits in different states. A user page becomes a kernel page when the kernel read/write it ①. Afterward, if user threads read this page, our page fault handler changes it back to user-mode with a read-only permit ②. This page is again capable of triggering a SMAP exception if the kernel reaccesses it ③. Our page handler suspends any user thread that tries to write a protected page ④. It then calls a sleep function, letting the operating system, and wakes up periodically to check the page's status ⑤. When the current system call ends, this page is restored to user-mode with its original permits ⑥. The write thread is also released and re-execute the faulting instruction to write the page ⑦. However, if the thread waits too long, it will be terminated to avoid a deadlock ⑧.	74
4.7. Bit 0 (Present): 0 indicates an invalid page. U/S (User/Supervisor): 0 user-mode accesses are not allowed to the page referenced by this entry. R/W : 0 writes are not allowed to the page.	74

- 4.8. ① User thread 0 invokes a system call with parameters. ② The kernel fetches one of the parameters from userspace hence triggers a SMAP exception. ③ Our page fault handler converts the page into kernel-mode to protect it. ④ User thread 1 tries to read the protected page and triggers an exception due to privilege violation. ⑤ Again, our page fault handler processes the exception and converts the protected page back to user-mode with a read-only permit. ⑥ The faulting instruction re-execute, and the user thread successfully read the data without knowing the previous exception. 75
- 4.9. To solve this, we choose to delay the write operation. ① User thread 0 invokes a system call with parameters. ② The kernel fetches one of the parameters from userspace hence triggers a SMAP exception. ③ Our page fault handler converts the page into kernel-mode to protect it. ④ User thread 1 tries to **write** the protected page and triggers an exception due to privilege violation. ⑤ This time, our page fault handler suspends the thread by calling a sleep function, letting the operating system schedule. It rechecks the page's status periodically when it wakes up. ⑥ When the current system call ends, the mitigation releases all the protected pages related to this thread. ⑦ Meantime, the page fault handler wakes up and realize that the page is released. Therefore it finishes the exception by executing the faulting instruction again. 76
- 4.10. Operations on the CR3 register are the decisive characteristic of process context switching, which triggers VM exit by default. By setting the SMAP bit in the CR4 image of the VMCS.GuestArea, it updates the CPU CR4 when the hypervisor enters the virtual machine again. Therefore, it is possible to use the hypervisor to enable the SMAP feature when a specific process is running on the CPU. 78

4.11. Page Fault Error Code. [68] P: 0 The fault was caused by a non-present page; 1 the fault was caused by a page-level protection violation. W/R: 0 The access causing the fault was a read; 1 the fault causing the fault was a write. U/S: 0 A supervisor-mode access caused the fault; 1 A user-mode access caused the fault. Notice , there is no exact error code regarding SMAP. In the context of an SMAP exception, the U/S bit is zero, which indicates kernel-mode access. We still need to combine this information with the faulting address and CS segment register to confirm the cause.	80
4.12. Left: PTE structure defined in C. Right: C code for changing one bit in the PTE structure, the corresponding assembly code generated by Microsoft C/C++ compiler, and the atomic instructions that serve the same purpose.	82
4.13. Performance overhead on the SPEC benchmarks incurred by the runtime load hypervisor. HVCi represent the Windows 10 native hypervisor for Hypervisor-Protected Code Integrity. All overheads are normalized to the unprotected system running benchmark.	85
4.14. Performance overhead in non-trivial applications. Overhead mostly being introduced on system calls that need to fetch user parameters	85
4.15. Pseudo-code for validating a file handler.	87
4.16. The attack thread tries to flip the data within the attack window, between instruction ② and ③. The kernel first touches the page at instruction ①, then the mitigation protects it afterward until the end of the current system call, creating a larger page-protect-window. As soon as the attack thread writes the protected page, the page fault handler suspends it until the protection ends.	89

4.17.	The family of the 27 vulnerabilities in the win32k module is mostly around the parameter security check. The checking code with <code>_W32UserProbeAddress</code> makes sure that the parameter is from userspace. It is a classic time-of-check-to-time-of-use. When checking, the value is benign where <code>[exc+8]</code> is less than <code>_W32UserProbeAddress</code> . After that, the attack can replace it with a malicious one before the second read.	90
5.1.	JTAG TAP state machine.	98
5.2.	The debug port (DP) receives host signals and maintains the JTAG state machine. The instruction and data are sent to the DAP core through IR and DR, respectively. Unlike the JTAG standard that needs to halt the processor before reading registers, using CoreSight DAP, the registers, SARM, and MMIO can be accessed during runtime without halting the processor.	99
5.3.	In the PLC manufacturer factory or during the product shipment, numerous employees can access the PLCs. The hardware backdoor installation should follow specific procedures to be efficiently accomplished without advanced software knowledge. The hardware backdoor can communicate with the attacker through the GSM network. Therefore it does not need to join the ethernet used by the ICS system.	102
5.4.	JTAG pins on the real-time microcontroller board. A square pad with ten pins is very likely to be a JTAG interface. We need at least to have the TDI, TDO, TMS, and CLK. RST is optional.	103
5.5.	Allen-Bradley 1769-L18ER-BB1B/B CompactLogix 5370 PLC. A: Power supply module B: Communication module C: Real-time module D: (16) DC Digital Outputs & (16) DC Digital Inputs Connector E: LED module	107
5.6.	A code snippet that calls <code>ROM_GPIOPinRead()</code> . The parameters are passed through registers. In this case, the <code>ROM_GPIOPinRead()</code> has two parameters. R0 contains the GPIO port address, and R1 indicates the pins to operate.	109

5.7.	Writing a byte of 0xF0 to address GPIODATA + 0x90. The bitmask only allows bit2 and bit5 to be modified. Therefore, only two bits are valid for the write operation. <code>u</code> indicates the new bit is ignored.	111
5.8.	The address GPIODATA+0x3C0 reads the high four bits of the GPIO port. The rest reads zero regardless of the actual value.	111
5.9.	Through wiring tracing, we find that the AT45DB21E SPI Flash Memory connects to LM3S2793's SSI0 interface. The eight solder joints on the left may be used to install four LEDs. They are controlled by GPIO port D, but they are not installed.	112
5.10.	The pseudo-code to initialize the I2C IO expander is reversed-engineered from the firmware. To make it more intuitive, we use some macro definitions as parameters. These macros' actual value is not difficult to find in the header files released by the vendor. The code is provided to help understand how to control the LED lights on the PLC through the I2C bus.	114
5.11.	Each pin in the Output register controls an LED light separately, so two 0xFF bytes can control 16 LEDs. If we only want to control a few of the LED lights in one register, we can call <code>ROM_I2CMasterControl()</code> with parameter <code>I2C_MASTER_CMD_SINGLE_SEND</code>	115
5.12.	The real-time module reads the input signals, runs the ladder logic, and then drives the output signal according to the result. Therefore this module needs to be connected to all other modules. In addition to the microcontroller and the flash memory, there are power regulators and IO chips on this module. The optically coupled isolators prevent high voltages from affecting the microcontroller when receiving the signal, and the power switches provide the electrical connection between the output pin and the voltage source.	116

5.13. The connector between the communication board (module B) and real-time board (module C). There are many pins, but the majority of them are power and ground. The microcontroller controls the LED module through the P607 connector. The communication module can block the operation of the entire real-time module through pin28. However, there are still a few pins that we have not figured out their functions. We consider it as one of our further works.	117
5.14. The power can be drawn from the JTAG pad or directly from the border connector p607. Since the JTAG pad provides the 3 volts source, we also need a DC-DC voltage step-up module that boosts from 3 volts to 5 volts. One GPIO pin from Teensy connects the PWR pin of the SIM800C board. It is used to deliver the power and reset sequence. After SIM800C initialized, the Teensy module sends AT commands through the serial port.	118
5.15. Both of the two registers are 35 bits long and can be scanned in/out through the JTAG protocol with IR instruction b1010 and b1010, respectively. RnW takes one bit, and zero indicates a write. A[3:2] selects the register within a bank.	119
5.16. Pseudo-code for memory writing through MEM-AP. Notice, CSW, TAR and DRW are all in the bank zero. However, the bank is specified in the DP.SELECT instead of a register in AP.	120
5.17. The Allen Bradley 1769-L18ER-BBIB CompactLogix 5370 PLC is in a 9.2cm x 13cm rectangle shape with sufficient space to contain the two bords and extra wires and connectors. The SIM800C HAT takes a full-size sim card, and the antenna takes much space.	121
5.18. We cover the necessary parts using tape to prevent a short circuit. The HARDDOOR connects to the PLC's microcontroller board's JTAG pad through a 10-pin socket and a long cable. The antenna for receiving GSM signal is also stuffed inside the PLC.	122

5.19. The hardware implant does not consume a lot of power, and the power consumption will only increase slightly when starting and executing the attack command. Sub-figure 1 shows the power consumption during the startup. Sub-figure 2.a shows the power consumption during an demo attack. 2.b indicates an output pin of the attacked PLC.	123
5.20. The two figures are the Allen Bradley 1769-L18ER-BBIB CompactLogix 5370 PLC's front and back view, in which both the PLC on the right has the hardware implant installed. There is no significant difference from the appearance point of view, and no parts are exposed to the outside. .	124
5.21. To accumulate execution time, we use a counter in the ladder logic and use an output signal to indicate the begin/end time of the test. During the standby test, HARDDOOR has attached the PLC, but no command is sent. We alter the PLC's output every 500ms to imitate a malicious operation for the attack test.	124
5.22. The GSM network may be congested when there are many users around. The large piece of message, such as 800 bytes, will be segmented into several packets, and the transmission time varies, but still, the received order and the message content are correct.	126

Chapter 1

Introduction

Software vulnerabilities are still the most critical aspect of system security. The author observes that many types of vulnerability have one characteristic in common. That is, system memory modification is a necessary part of exploitation. The attacker may overwrite data structures, reoccupy freed buffer, or repeatedly write the same address to exploit the vulnerability. Therefore, the protection of system memory is also necessary and plays an essential part in system protection. This dissertation aims to solve several state-of-the-art vulnerabilities challenges and provides systematic mitigation solutions for each. Moreover, it presents a new type of attack that deploys a small-size parasitical hardware implant to control an embedded device such as a PLC, breaking existing software detections and protections.

In Chapter 2, the author aims at mitigating heap overflow vulnerabilities. Some vulnerabilities try to gain control of the system by first compromising system metadata. Typically, in a heap overflow attack, the most leveraged metadata is the heap metadata, the heap management's allocation records. In 2015, the Linux Glibc library's severe weakness, so-called the Ghost vulnerability, is of this type. It takes advantage of heap metadata to obtain complete control of the victim system. The author argues that the issue with system metadata is as follows. System metadata is essential to be protected. However, instead of integrated with the kernel, the metadata is left with user code. The reasons are various. Making less privileged code in the kernel is a significant one. Therefore, the author presents a novel heap allocator that protects heap metadata with a secure environment in the user space, using the hardware feature Intel Software Guard Extensions (SGX). SGX provides separated memory spaces, so-called enclaves, to protect critical user data. With the heap metadata reorganized and protected, the

mitigation prevents vulnerabilities from abusing metadata, hence stop attacks of such kind.

This work also solves the challenge of exposed critical data using SGX enclaves. It is also fatal to system security if the attack reads critical data such as a private key. The notorious Heartbleed vulnerability, in essence, is an information leak vulnerability that widely affects web servers that use the OpenSSL library for stream encryptions. The RSA private key exposed in the server's process memory leaked when the attacker illegally read the memory cross-boundary. This work stores the private key and encrypting/decrypting routines inside an SGX enclave to only send out decrypted messages. The author evaluates the key's security using a Metasploit module for the Heartbleed exploit and shows that an attacker cannot access the key despite a vulnerability.

In Chapter 3, the author aims at mitigating use-after-free (UAF) vulnerabilities. UAF exploits have contributed to many software memory corruption attacks in recent practices. They are especially popular in the world of web browsers. The competition between exploit and mitigation becomes a cutthroat arms race. Despite many successful UAF exploits against widely-used applications, state-of-the-art defense mechanisms have proved to leave the systems vulnerable still. This author argues that a successful UAF exploit is feasible because of the fine-grained determinism provided by existing heap memory allocators. Namely, the state-of-the-art exploit uses the in-object member fields of a particular object to refill the victim object's function table pointer. This work introduces a new defense strategy that leverages additional memory buffers to make allocation outcomes locally unpredictable to adversaries. This fine-grained non-determinism prevents exact alignment of subsequent allocations and in-object member fields. It significantly lowers the success rate of a UAF exploit even in the presence of heap sprays. The author also validated the defense using real recent UAF exploits against several CVE vulnerabilities in large and popular software packages (Firefox and Tor browsers). The defense can terminate all the exploits in the early stages and prevent the gadget addresses successfully located for the intrusion's follow-up return-oriented programming steps. Its runtime performance overhead was negligible (1.2

In Chapter 4, the author aims at mitigating kernel-level time-of-check-to-time-of-use

(TOCTOU) vulnerabilities. Kernel-level TOCTOU widely exists in operating systems, especially Microsoft Windows. When serving a system call, the kernel inevitably gets parameters from the userspace. Read the same user-mode variable repeatedly may lead to data inconsistency under a race condition between the kernel and userspace. The author notices the Windows graphical subsystem kernel module couples with user-mode libraries, and it accesses user-mode data structures loosely, among which double fetches on the same address are not unusual. To find the bugs with the typical memory-access pattern, the author develops a fuzzing tool that effectively finds kernel-level TOCTOU candidates.

Furthermore, to mitigate such vulnerabilities, this work is the first runtime mitigation technique on Windows. It does not require Windows kernel source code or modifying kernel binary. The core of this mitigation is to use Supervisor Mode Access Prevention (SMAP), a hardware feature, to detect kernel access to userspace data. To leverage SMAP, the author makes the kernel adaptable to recover from the fatal SMAP exceptions. Due to the Windows system's complex nature, the author further develops a lightweight hypervisor to contain the system-wide hardware feature SMAP into specific processes to prevent deadlock caused by nested SMAP exceptions. The hypervisor also improves the flexibility and performance for the mitigation. The author evaluates this mitigation and the lightweight hypervisor with 18 benchmark programs and real-world applications. The results show that the mitigation imposes little extra overhead (less than 10% on average).

In Chapter 5 the author presents a new type of attack. Nowadays, critical infrastructure such as the power grid is vital to national security. Their failure or incapacity would have a significant impact on people's daily life on a large scale. However, they are under emerging advanced persistent threat (APT) attacks since the infrastructures systems are automated and computer-controlled. The programmable logic controllers (PLCs) are the neurons that control the physical system. In most APT attacks, a stealthy backdoor usually is the core that allows the attacker to hide in the dark without being detected and launch remote malicious operations at a particular moment. However, to achieve further stealthiness and bypass existed software mitigations, it needs to evolve

from high-level software into low-level hardware.

The author presents a small-size parasitical hardware implant that attaches to the PLC's circuit board. It controls the PLC by modifying the digital signal or hijacking the various buses on the boards. This attack can be deployed either during the supply chain or stealthily installed in remote plants. The hardware implant contains a cellular chip that provides a remote control channel, which allows the attacker to organize a multi-point distributed attack.

The author implements and evaluates this hardware implant with widely deployed Allen Bradley PLCs. The result shows that such a hardware backdoor does not change the firmware or induce overhead to the system, thus no integrity violation. It can secretly change the PLC's output without showing any trace. Furthermore, the attacker can even penetrate air-gapped networks communicating with it and conduct a simultaneous attack with multiple controlled nodes.

Chapter 2

Dynamic Memory Protection via Intel SGX-Supported Heap Allocation

1 Introduction

Heap buffer overflow vulnerabilities are increasingly becoming a priority for security researchers. Critical vulnerabilities such as CVE-2015-0235 [130], known as *GHOST*, have provided attackers the ability to exploit heap buffer overflows by exploiting holes in the current heap protection mechanisms. A popular perspective among security experts is that the vulnerable program should take full responsibility for the buffer overflow. However, after our investigation, we found out that the lack of full protection of the program’s metadata also accounts for the software corruptions as the metadata, which keeps track of heap management information, is located in memory without being protected and/or isolated properly. Although metadata is essential for heap operations of applications, it is not supposed to be directly accessible from the user applications. The only legitimate parties that can access and maintain the metadata should be heap functions. Unfortunately, this rule is frequently violated and leaves the programs vulnerable to exploits. In response, there have been efforts to address this issue [36][79][9][60]. Although there have been mitigation attempts, e.g., integrity checking and encryption, the metadata is still left partially protected in memory.

Recent works have focused on bypassing these mitigation techniques that attempt to secure the critical metadata. [178] proposed an approach to circumvent the Internet Explorer (IE) browser’s mitigation provisions, such as Data Execution Prevention (DEP[6]), Address Space Layout Randomization (ASLR[156]), as well as any checks for Return-Oriented Programming (ROP[136][146]) gadgets. The security level of the

IE script engine is determined by the parameter `SafetyOption`[178], which is a critical variable controlling the execution privilege of the IE script engine. If an attacker has access to arbitrary memory space of a targeted process, e.g., through an arbitrary memory writing vulnerability, then this `SafetyOption` variable can be modified to allow execution of malicious scripts with escalated privilege. In order to protect the integrity of the `SafetyOption`, Microsoft published a patch for the JScript engine with an added function `ScriptEngine::GetSafetyOptions`, which generates a 32 bytes hash digest binding the `SafetyOption` value with the execution context. Hence, if the adversary modifies the `SafetyOption` directly in memory without calling the `GetSafetyOptions` function, the tampering operation will be identified by the hash digest. However, this solution is not a panacea as an adversary may still be able to modify the `SafetyOption` with access to other object data and the corresponding hash digest in memory.

In addition to protection of metadata, we cannot ignore the existing need to protect critical data that is exposed from normal vulnerabilities. For example, the Heartbleed vulnerability [46] can be exploited to extract the private key of a web server. The vulnerability allows an attacker to read a large portion of memory (65535 bytes). This can lead to the exposure of critical data, such as an RSA private key. Even if the vulnerabilities are patched, mechanisms should be in place to protect the critical data of a user program in spite of zero-day exploits.

The aforementioned vulnerabilities emphasize the core safety issue: critical data cannot be secured by security solutions if the memory can be arbitrarily modified. As a consequence of arbitrary memory reading and writing vulnerabilities, the sensitive data might eventually be tracked down. Therefore, more reliable solutions must be implemented that cannot be corrupted by an adversary. Technology such as the *SGX (Software Guard Extension)*[104][71][5] has been provided as a powerful tool for the protection of critical metadata. SGX enables the use of *enclaves*, which are hardware-separated memory spaces that protect user mode code. The memory within an enclave is hidden from illegitimate users, including users with higher privileges. The enclave provides a safe location to hold the critical user data and/or program metadata so as to eliminate arbitrary memory access. The only legitimate accesses to the enclave will be

provided by carefully designed interfaces that hide specific operations and only return results.

In this paper, we introduce METASAFE, a new suite of heap APIs based upon SGX and its enclaves that provides simple and efficient heap buffer management functionality so as to store the metadata inside an enclave. With our tool, if a heap buffer overflow happens, the heap metadata cannot be affected. We also integrate the concept of clean separation of metadata from user buffers from *jemalloc*—which is an excellent dynamic memory allocator in FreeBSD and NetBSD—into our design. Because the metadata is independent of other program data, it can be relocated to any other position of the process memory space. Therefore, the metadata is placed in an enclave to ensure its protection. In addition to the protection of critical metadata, we present a set of user-friendly APIs so that a user program can easily access the program’s critical data, e.g., a web server’s private key, that is protected by an enclave.

Our implementation of heap APIs consists of two major components: the *SGX* enclave, which is responsible for maintaining heap buffer record, and a memory management interface that sits outside of the enclave and provides an interface for allocating virtual pages from the operating system. These two components work together to coordinate virtual page allocation. The enclave does not handle virtual page operations as they involve communication with the operating system. Instead, the enclave informs the memory management interface to allocate virtual pages from the operating system. After, the memory management interface sends the information of where the pages are and how long the range is back to the enclave. With this page information stored as metadata inside the enclave, routines inside the enclave are able to fulfill the user memory allocation request by calculating the exact user buffer location.

Contributions. METASAFE enables protection of a program’s critical metadata by leveraging the hardware-level memory separation through SGX enclaves. The technical contributions of this paper are as follows: *i)* We provide a set of new heap management APIs that leverage Intel SGX to protect heap metadata from being tampered with by heap buffer overflows. The critical metadata is protected by being placed in SGX enclaves. *ii)* We implement the aforementioned suite of APIs to also allow user

programs to easily manage access to and storage of critical data in SGX enclaves. *iii)* We evaluate METASAFE by integrating the APIs into several widely used open source projects, including nginx, nullhttpd, 7zip, netcat and gzip. *iv)* We further evaluate the METASAFE APIs by implementing OpenSSL using the SGX enclaves. We use a Metasploit module to demonstrate the effectiveness of METASAFE against the Heartbleed exploit.

2 Background and Case Studies

Modern operating systems are typically isolated from user applications. For example, in x86 systems, the code and the data of the operating system own the highest privilege *ring0*, while the user-level applications work with the lowest privilege *ring3*. The user applications can not modify the code nor the data of the operating system without going through specific interfaces of the operating system. The user applications can still effectively utilize system resources without knowing the details of the underlying implementations, e.g., the specific data structures and related data. This philosophy of isolation and modularization guarantees the robustness of the operating system: the crash or compromise of a user application does not affect the normal operations of the operating system. The same philosophy can also be partially applied to user mode programs. Because services provided by OS user mode components such as C runtime libraries are also considered system services, they are also isolated and protected from user programs.

Although these system services and user mode services isolate their underlying data structures from user programs, their associated metadata is typically not isolated. The user programs do not necessarily need access to this metadata. For example, when a user application applies for a virtual page, the operating system only returns the address of the allocated virtual page to the user application, while the underlying operations, such as associating the physical memory page with the virtual address space, are constrained within the operating system. A user program on Windows does not need to know that the operating system maintains the virtual address space of a user application using a red-black tree data structure. Similarly, isolation mechanisms should be used to protect

all code and data embedded in the kernel as user processes should have access to the kernel only through system calls.

2.1 Heap Metadata Vulnerabilities

The heap used by user applications can be considered a system service that utilizes standard interfaces to provide services to user mode programs. For example, a user program that calls the *malloc()* function to allocate a block of memory is only concerned with the buffer address returned by the *malloc()* function. The user program does not need to know how the *malloc()* function implements the memory allocation nor the type of data structure being used to keep track of the allocated buffer. However, an important difference between user applications and the kernel is that the metadata for the heap of user applications does not receive any protection from isolation. Hence, a buffer overflow in a user application may overwrite the heap metadata, leading to severe security concerns for the user program. Another example of this kind is that the metadata of the Windows exception handler (SEH)[50][154][153][102] includes a code pointer. Once a buffer overflow occurs and overwrites the code pointer, the related process might be maliciously modified. Even though new mitigation and protection methods have been introduced to protect the heap and the metadata of the SEH, attackers continuously find new vulnerabilities to bypass the protection mechanisms. Therefore, we will introduce how SGX can be leveraged to implement isolated protection regions enabled by the hardware to protect the aforementioned metadata.

Heap Buffer Overflow Attacks. Heap metadata is typically exploited using two kinds of heap buffer overflow attacks. The first attack implements a buffer overflow that over-writes just the heap metadata in order to induce arbitrary memory writing [36][160][7]. This vulnerability is prevalent especially on *dlmalloc* and its derivatives. Because this attack was discovered over a decade ago, many protections have since been implemented, e.g., integrity checking before linking or un-linking a list. These protections have significantly increased the difficulty of overwriting metadata alone to achieve an exploit.

The other attack first overwrites metadata in order to make the allocator behave


```

struct malloc_chunk {
    // size of previous chunk (if free)
    INTERNAL_SIZE_T prev_size;
    // size in bytes, including overhead
    INTERNAL_SIZE_T size;
    //double links -- used only if free
    struct malloc_chunk* fd;
    struct malloc_chunk* bk;

    // Only used for large blocks:
    // pointer to next larger size
    // double links -- used only if free
    struct malloc_chunk* fd_nextsize;
    struct malloc_chunk* bk_nextsize;
};

```

Figure 2.1: `malloc_chunk` structure

abnormally[130][159][9][8]. This allows the attacker to further control other areas of memory, including critical data of the vulnerable program. A case study will now be presented to exemplify the latter attack.

Case Study. In 2015, a notorious vulnerability “GHOST ” [130] infected a large variety of software by exploiting `__nss_hostname_digits_dots()` function in the GNU C Library (glibc). This bug existed in the `__nss_hostname_digits_dots()` function and could be triggered either locally or remotely via the `gethostbyname()` function. It is a heap buffer overflow attack in which the attacker injects certain malicious input exceeding the length of the heap buffer. In this case, a buffer overflow of just 4 bytes or 8 bytes can be implemented depending on the architecture of the operating system (32bit or 64bit). These 4 or 8 bytes are sufficient for corrupting the heap metadata, which is located immediately after the heap buffer.

Figure 2.1 presents the structure of the metadata located immediately following the buffer that belongs to the next free chunk. The `prev_size` and `size` fields are both 2 bytes long. These are the variables that are controlled by the attacker in the GHOST vulnerability.

Figure 2.2 highlights the overflow of the data structure as well as what specific variables will be corrupted. The attacker has comparatively limited capability as he

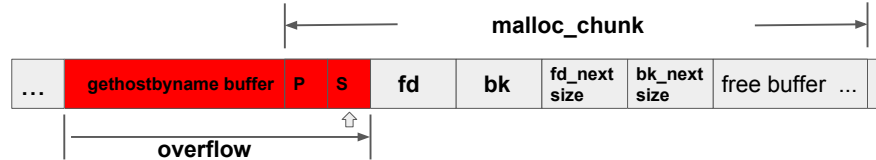


Figure 2.2: With only 4 or 8 bytes overflowing, it is enough to overwrite the size field of next free chunk, which will enlarge the next free chunk. The arrow indicates where the metadata corruption happens

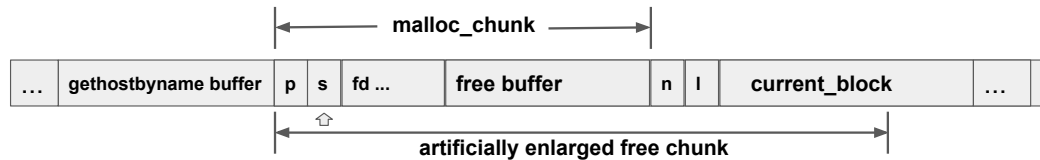


Figure 2.3: After attacker overwriting size field of the adjacent free chunk. When next time calling `malloc()`, the program get a over large buffer which contains part of other buffer. In GHOST's case, that part belongs to an internal memory allocator and could be further exploited.

can only manipulate either 4 byte or 8 byte memory blocks of the metadata. However, in some scenarios, that is powerful enough to further make a critical exploit. It manipulates the size field of the next free chunk, making the buffer larger than it should be. Subsequently, when the `malloc()` function is called again, it will allocate an artificially enlarged buffer that includes some memory belonging to the next buffer. As a result, the newly allocated data block might overwrite other data structures in memory. Figure 2.3 shows the sequence of data structures after the attack has been implemented. This vulnerability only needs to overwrite a very short range of memory. It epitomizes the need to secure the metadata as it is become a key target for buffer overflow attacks.

2.2 Non-Control Flow Attacks

Control flow attacks are some of the most prevalent attacks in recent decades. Control flow attacks target code pointers so that the adversary can modify the pointers and hijack the program flow directly. As a countermeasure, effective mitigation mechanisms have been proposed to protect against control flow attacks, and hence raise the bar for attackers to launch the control flow attacks. Consequently, adversaries recently

turned their attention to non-control flow data, which is the critical non-pointer data that indirectly determines the program flow as well as influences system/program level settings [29]. For example, non-control flow data can be the variables that control the system security settings, or the crucial flags that determine the control flow of a program. These types of attacks are known as *non-control flow* attacks. These attacks have been used to implement exploits for severe vulnerabilities, such as those presented in [178] and [170]. We will focus on the former attack to exemplify the critical nature of these vulnerabilities.

Case Study. One example of a non-control flow data attack is the recent attack that targeted the `SafetyOption` of the IE browser. In 2014, researchers disclosed an attack against the IE 11 by exploiting one arbitrary memory writing vulnerability without modifying any code pointers [178]. Traditionally, an attacker has to leverage such a vulnerability by overwriting certain code pointers first. Later, when this code pointer is called, the attacker assembles Return Oriented Programming (ROP) gadgets to bypass all the defense mechanisms, such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) and other ROP attack detection and mitigation methods. As the countermeasures against this type of attacks have become sophisticated, it has been much more difficult, if not feasible at all, for attackers to overcome all the prevention mechanisms. Alternatively, another attack approach was discovered: a global variable, `SafetyOption`, is vulnerable in the sense that it can be modified by any user with the capability of arbitrary memory writing. Non-control flow data enables an adversary with arbitrary memory writing capability to simply find and modify the `SafetyOption` directly, without overcoming most of the prevention mechanisms, if not all. The catch is that this attack against the `SafetyOption` takes advantage of privilege settings, and therefore grants an adversary the highest privilege.

The same kind of attacks are becoming more and more pervasive[170][27], and a new field of exploitation has been opened. The underlying root cause is that every component of a program has the same capability of accessing the program's whole memory space without restriction nor verification.

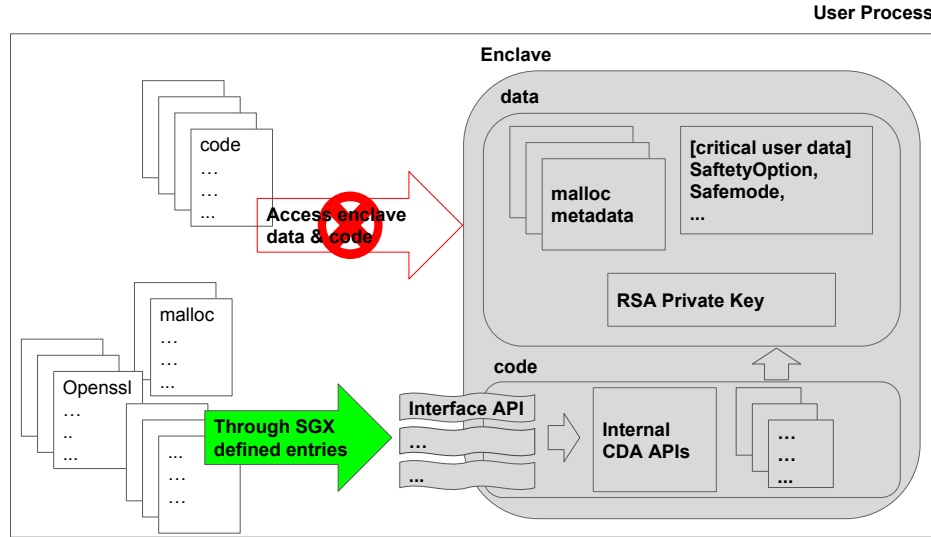


Figure 2.4: High-Level Overview of METASAFE

3 MetaSafe Overview

Figure 2.4 provides an overview of METASAFE’s implementation. METASAFE’s main goal is to protect critical data using memory isolation within SGX enclaves. The SGX enclaves are used to store critical data that should not be accessible directly by user programs. In particular, METASAFE uses the enclaves to store heap buffer metadata that does not need to be accessed by user programs. The heap buffer metadata has been a target of recent buffer overflow exploits and should be considered critical data.

If a user program wishes to allocate a buffer in the heap, METASAFE provides a memory management interface that interfaces with internal critical data access (CDA) APIs located within the enclave. These CDA API’s can access and modify the critical data within the enclave while returning only the necessary results to the memory management interface communicating with the user programs. This restricts the user programs from indirectly modifying the critical metadata, whether the intentions were benevolent or malicious.

In addition to the memory management interface, METASAFE provides user programs with a suite of APIs that allows the programmer to explicitly protect data that is considered critical to the program. As we previously mentioned, an RSA private key is extremely critical data whose exposure can severely compromise a web server.

The API's have been implemented in a convenient fashion that does not require many changes to the original code base. This allows the programmers to interface with the CDA APIs to protect any specified critical data of a user program.

4 Threat Model

This section presents METASAFE's threat model. We assume a Windows operating system since currently only Windows supports the SGX SDK—an essential component of our solution. A process of a user application, which runs in the Windows operating system, has a heap buffer overflow vulnerability or an arbitrary memory access vulnerability. This user application might employ DEP, ASLR and/or other ROP mitigation methods to defend against exploits. However, the adversary has the capability of arbitrary memory reading and writing in order to tamper with any data in the data section of the process. The adversary also has the capability of calling `malloc()` and `free()` as many times as needed to arrange the heap memory layout. Additionally, the attacker can overflow a heap buffer in order to overwrite adjacent memory. This attacker may also know how to pass malicious inputs so as to transgress the boundary of a heap buffer, exploiting the heap buffer overflow vulnerability in order to hijack the control flow for execution of arbitrary code. To protect against such a powerful adversary, we assume SGX is available, which provides a hardware-separated memory space for sensitive data/code.

5 Design

In this section we will provide an overview of the design of our critical data protection mechanisms that are implemented using SGX enclaves.

5.1 Program Isolation

The Software Guard Extensions (SGX) technology was developed by Intel to provide an isolated space for selected data and code storage and execution. In general, SGX could provide one or more isolated enclaves inside of a process. An enclave is an isolated

region of code and data which is protected by the processor’s hardware. Only code that runs within the enclave can access data within the same enclave. Furthermore, other parts of the program cannot execute the code within an enclave. Outside programs can only execute calls (ECALL) using the enclave interface, which is predefined using an Enclave Definition Language (EDL) file. EDL files describe enclave trusted and untrusted functions and types used in the function prototypes. The code that runs within the enclave can access data outside of the enclave, but it also cannot directly execute the code outside. This also needs to be predefined (OCALL) by an EDL file. This protection can prevent vulnerable code from running within an enclave from being attacked and taken over the whole outside program. Now that we have presented the general overview of SGX, we will briefly discuss individual components and similar technologies along with their differences.

The Enclave Page Cache (EPC)[73] of SGX is a protected area of memory used to store enclave pages. It’s part of DRAM, but cannot be accessed by other programs. It is also protected by an encryption engine, so when one enclave page is being swapped out to normal DRAM, it’s content is encrypted. The page is also 4KB aligned. Each page is either free or belongs to one enclave. The security attributes of EPC pages are stored in the Enclave Page Cache Map (EPCM). Each entry represents one EPC page and includes information such as the validity accessibility of the page, which enclave owns the page, the type of the page, read/write/execute permissions of the page, as well as the linear address through which the enclave is allowed to access the page.

A few new instructions are needed to support SGX. Some are privilege instructions, i.e., EINIT to initialize an enclave, EADD to add a page into enclave, and EEXTEND to measure a page. The rest of the new instructions are user mode instructions, i.e., EENTER for calling a function inside an enclave, EEXIT to allow code running inside enclave to exit to the normal world, as well as ERESUME to allow the code to reenter the enclave.

SGX is a the successor of Intel’s Trusted Execution Technology (TXT)[168]. TXT similarly provides a secure environment, but it has to be either at the boot stage of a system or when the entire system is paused. It resets the CPU and runs into a secure

environment while auditing every piece of code that runs subsequently. This solution is not very convenient for user mode programs.

Another isolation solution is the ARM TrustZone[3]. This solution provides two separate worlds for code storage and execution: a normal world and a secure world. Code running inside the secure world has a higher privilege and can access the physical memory space of the normal world, while the code running in the normal world cannot access the memory of the secure world. The two worlds communicate through a privileged mode in the ARM CPU called Monitor mode. You can run two operating systems simultaneously in two worlds.

SGX is more flexible in the sense that you can create many enclaves in one system, or even in one process. When the code inside one enclave is scheduled to run, other parts of the system don't freeze.

Processes can be easily used as a mechanism to provide isolation. Code running within one process uses inter-process communication to access other processes' data. The operating system provides several mechanisms for inter-process communication, such as shared memory, socket communication, and named (or unnamed) pipes. But if every system mechanism needs to isolate data through multiple processes, it would be too many processes to maintain for one single program. Each additional process will require extra system resources, such as page tables. This obviously is not an optimal design. We evaluated the performance for the most commonly used inter-process communication methods for data isolation in the evaluation section.

The system/user mode architecture also provides isolation. However, x86 architecture only provides one kernel mode and three user modes. Furthermore, most operating system implementations include only one kernel and one user mode. In addition to these limitations, bringing isolated data into kernel mode is impractical as the kernel mode cannot provide fine-grained isolation for each process. Since the kernel mode has a higher privilege. Any additional code and/or data would increase the possibility that the system being compromised.

5.2 Secure Heap Management

One decade ago, computer memory scarcity was the main concern when implementing memory allocators. Intuitively the allocator gives the exact amount of memory for a buffer that the user requested, and a free list is used to manage the buffers. Free lists are essentially doubly linked lists and are a common data structure used by allocators such as `dlmalloc` and the Windows memory allocators. The doubly linked list and other information such as the size of the buffer is considered metadata. The memory is used efficiently and the adjacent free buffers could be merged in order to satisfy larger allocation requests and reduce fragmentation. Since buffers including metadata are next to each other, the drawback is obvious. Once there is a buffer overflow, metadata could be easily overwritten. An attacker could construct special allocate and free requests with overflowed data in order to manipulate the metadata for an exploit.

Jemalloc has a different design. There are many fixed-size regions¹ in jemalloc. Regions are stored and managed by a *run*. One run is just a continuous piece of memory with the same-sized regions placed adjacently in memory. There is no metadata between the two regions (although there could be several bytes used as a red zone, those are not considered as metadata). Because there are no link pointers there, the attacker can only modify adjacent buffers with a buffer overflow—a different attack that is not considered in our paper. The aforementioned attacks against `dlmalloc` will not succeed in this case.

Although jemalloc protects against this particular attack, metadata still exists in several components of jemalloc, such as the run header and the chunk header. The attacker can still manipulate the memory allocation pattern in memory and leverage heap buffer overflows to overwrite these components[9]. This metadata exists between buffers or within the safe page of other buffers and is especially vulnerable to buffer overflows. In the x86 architecture, memory protection can only be applied in a page granularity, so you cannot simply separate the metadata by putting guard pages[106] between them. This solution would not be practical as it would require a vast amount

¹Jemalloc refers to buffers as "regions"

of memory overhead (4KB per page). In order to protect the metadata, it should be separated from user buffers. Therefore, we will extend concepts of jemalloc. All metadata is gathered together by design and separated from the user buffers, such as the run header and the chunk header. This metadata can exist independently at any location in memory.

Additionally, the integrity of the metadata will need to be verified. There have been many techniques to protect metadata from buffer overflows, e.g., the Windows 8 heap allocator uses a guard page to isolate metadata from user buffers. Furthermore, solutions have been implemented on the x86 platform that use segment registers to isolate a range of memory [87]. However, these segment protections are not available on x86-64 platforms, which is too important to ignore nowadays. Although there have been other integrity checking and metadata encryption solutions to prevent malicious tampering of the metadata, these solution still leave metadata in a place where attackers can reach. It is impractical to verify or encrypt all the metadata every time there is a memory allocation request. We will discuss more of these mechanisms in the Section 8.

We propose a heap implementation called `sgx_malloc()` that leverages the enclave capabilities of Intel’s SGX. Our design further proposes to have the metadata separated in memory by hardware. The metadata will be saved inside an enclave with all operations on metadata running inside the enclave as well. Functions such as `malloc()` and `free()` are divided into an internal portion located within the enclave and an external memory management interface. The memory management interface is responsible for interfacing with the user programs and requesting memory pages from the operating system. The internal portion runs inside the enclave and is responsible for metadata related operations such as calculating the address of each allocation or a buffer freeing request. Since there is absolutely no metadata exposed in the user memory space, an attacker cannot corrupt the metadata using the aforementioned vulnerabilities.

Several concepts from jemalloc have been adopted. We use almost the same terminology, such as *region*, *run* and *bin*, to compose the metadata of `sgx_malloc()`. Similarly, we have arranged the memory into equally sized regions to form runs. Runs

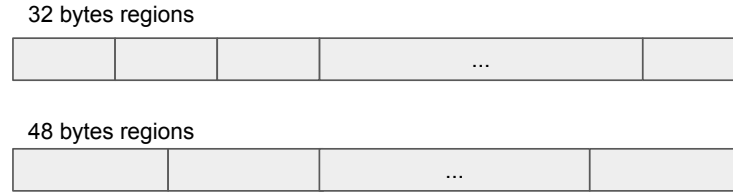


Figure 2.5: The same size region are within one run, regions are next to each other, no metadata between them

use allocation bitmaps to record which region is used. They have several predefined types, such as 8-byte, 16-byte and 32-byte runs. The size of a run corresponds to the size of the regions in a run, e.g., an 8-byte run has 8-byte sized regions. Figure 2.5 depicts how regions are laid out within runs. The number of regions contained in a run varies. The key difference between `jemalloc` and `sgx_malloc()` is that our solution stores metadata inside the enclave as opposed to inside a virtual page in memory. We also use bins to manage multiple runs of the same size. A bin is composed of a red-black binary tree to trace all the locations of the same size runs. This allows subsequent memory allocations to start at the lowest possible end of memory to reduce fragmentation. Figure 2.6 provides an overview of the `sgx_malloc()` metadata storage. Memory requests that larger than 1KB are considered large. We also use a red-black binary tree to manage such memory allocations.

The `sgx_malloc()` request is initially not aware of any page information for allocation. Therefore, the first `sgx_malloc()` request cannot be satisfied immediately. For example, if the user requests 20 bytes of memory to be allocated, the external memory management interface of `sgx_malloc()` will send a request to allocate 20 bytes while the inner portion, implemented as a separate function called `i_sgx_malloc()`, will find that no run has been created yet by checking `bin32`. Therefore, a new 32-byte run is needed. The `i_sgx_malloc()` function will return a value to notify the external memory management interface while providing the size of memory that is needed (in units of pages) and the type of run that is needed. The external memory management interface then requests pages from the operating system. Once the operating system has satisfied this request, the address of the allocated pages and the type of the run are both sent into enclave by calling the `i_newrun()` function. The `i_newrun()`

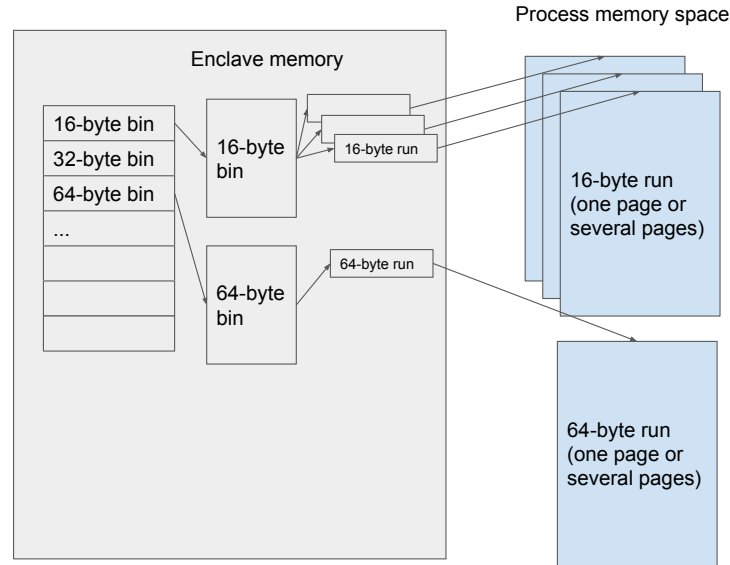


Figure 2.6: `sgx_malloc()` metadata is stored inside one enclave, in only contains the information that where the run is located in memory and which region in that run is used

function creates a new run of the specified type. The `sgx_malloc()` function will then call `i_sgx_malloc()` to complete the previous request, which can be satisfied given the newly created run. Through the existing run, the region address is calculated and returned to the memory management interface to be returned to the user program.

The process for `sgx_free()` is implemented similarly. After passing the buffer address into the enclave, the inner portion finds the right run by going through the corresponding bin. After a series of validity checks, the corresponding bit in the allocation bitmap of the run is set if the region is valid.

It is worth noting that heap metadata needs sufficient space to grow. In current design, we reserve enough memory pages in the enclave to store it. Although the current version of SGX (1.0) cannot dynamically extend the enclave memory space, later versions will have the ability to add more pages to an enclave at runtime[72].

5.3 Protection of User Program Critical Data

The aforementioned solution protects metadata and related functions by packaging them together into an enclave. We will leverage a similar means of protection that provides a mechanism to protect critical data of a user program.

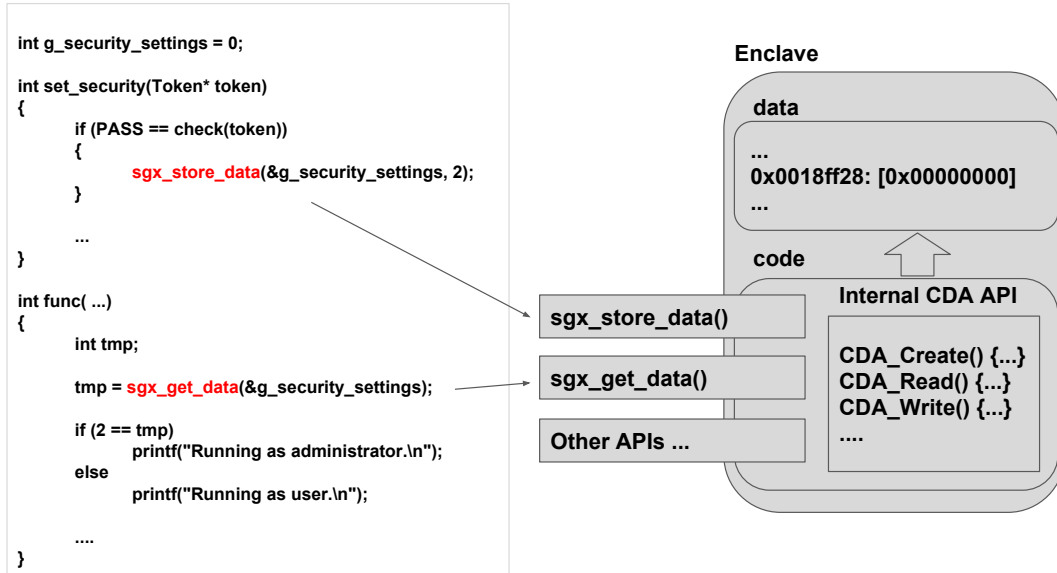


Figure 2.7: Memory layout of user program protection through enclave.

In this context, critical data refers to non-control flow data, i.e., data that is not a code pointer but can be used indirectly to corrupt a program. When an attacker hijacks a code pointer, in theory he can run arbitrary binary code. In practice, the attacker needs to find a way to bypass all the related protection mechanisms in addition to finding a way to place shellcode in a reachable location. Although it has become more and more difficult for an attacker to exploit fully protected software, we have presented a case study where the corruption of a single byte in memory can defeat all the security protections.

We used the enclave implementation to provide a safe location to store critical data and non-control flow data in the process memory address space. We use the data's address as an index for storage since the memory address is unique to a process. Inside the enclave, we use a red-black binary tree to improve search efficiency. When a process starts, we first create and initialize an enclave. When the data needs to be stored, one of the APIs is called to provide the data's address and value. The API will update the value at an address if the value has changed since the last call.

Figure 2.7 depicts how the corresponding metadata is stored inside an enclave. This application can prevent a typical DOP attack[64] which leverages the vulnerable local variable to achieve an arbitrary memory write attack. Essentially, in order to elevate

privileges, the DOP attack will modify global settings of the vulnerable program, e.g., the `SafetyOption` in the IE browser in the case study we previously presented. We consider this non-control data as critical data. Logically, such settings should not be accessible by an unrelated part of the program. In the case of the IE browser, the `SafetyOption` should not be accessible by the web content. Only the logically related functions need to call APIs to access it. In some sense, it binds the critical data and the member functions together. The critical settings should exist only in the enclave and registers. The ideal circumstance is that no temporary local variables are used and, hence, no memory leaks and tampering are possible. But in practice, it's difficult to assign a register exclusively to hold an variable in a high level programming language. Local variable is used in a short amount of time, but the attack surface is reduced significantly.

6 Metasafe Implementation

In this section we present the implementation details for our aforementioned solution. We propose two ways to protect data using an enclave. One requires that the data and its associated functions are closely integrated with the enclave. Data is stored in the enclave and there is no corresponding variable declared outside of the enclave. Data associated functions contain inner functions that work inside the enclave and outer functions called by user programs as an interface. Data and functions are packed together as an independent library, providing services to the user program.

Memory Management Interface. In order to implement the `sgx_malloc()` API, we offered a set of standard C runtime library functions use for memory allocations. Namely, we implement custom versions of `malloc`, `free`, `calloc`, `realloc` and `_msize`. Furthermore, we provided two more functions, `sgx_malloc_init()` and `sgx_malloc_uninit`, to initialize the creation of the enclave as well as to cleanup the resources of enclave when the program is finished, respectively.

We used Intel Software Guard Extensions Evaluation SDK for Windows OS to compile our `sgx_malloc()` library. The `sgx_malloc()` function that was implemented

to replace `malloc()`. Other functions such as `calloc()` and `realloc()` could be seen as its derivatives. As mentioned previously in Section 5, `sgx_malloc()` is divided into two parts. The inner part or trusted part is the set of functions that access the internal metadata. The `sgx_malloc()` function is the wrap of the enclave inner functions such as the `i_sgx_malloc()` and `i_sgx_newrun()` functions. These functions are in a stand alone library. Those are real enclave interfaces which will be defined through the EDL (Enclave Definition Language) file.

Furthermore, we implemented sample wrapper functions to interface with the critical data access API (CDA API) inside the enclave for user programs. Figure 2.7 depict a sample program that use the METASAFE APIs to store and retrieve critical data. The functions `sgx_store_data` and `sgx_get_data` are intended to be wrapper functions for storage and retrieval functions implemented by the user program. These sample APIs simply demonstrate that METASAFE's APIs can be extended beyond the protection of just heap metadata.

Critical Data Access Management. We created a common critical data access (CDA) broker mechanism to provide critical data in the SGX domain. The broker system includes three parts: 1) the critical data manager 2) the critical data access broker and 3) the critical data access stub. Both the critical data manager and the broker are running within SGX inside. The stub code is outside of SGX and provides the stub functions for a set of APIs.

The full list of the CDA components is as follows: **Critical Data Manager:** The critical data manager provides the functions for SGX memory allocation and deallocations, reading and writing data, as well as encryption/decryption. **Critical Data Access Broker:** The critical data access broker defines the broker request command for a stub to use. **Critical Data Access Stub:** The critical data stub provides a set of APIs to the application for critical data isolation usage. Within the APIs implementation, it creates broker request per API.

The system also provides a set of APIs for importing and exporting data, creating and destroying critical data, reading and writing critical data, as well as encryption/decryption. The list of APIs is as follows: **cda_import:** The import API can be used

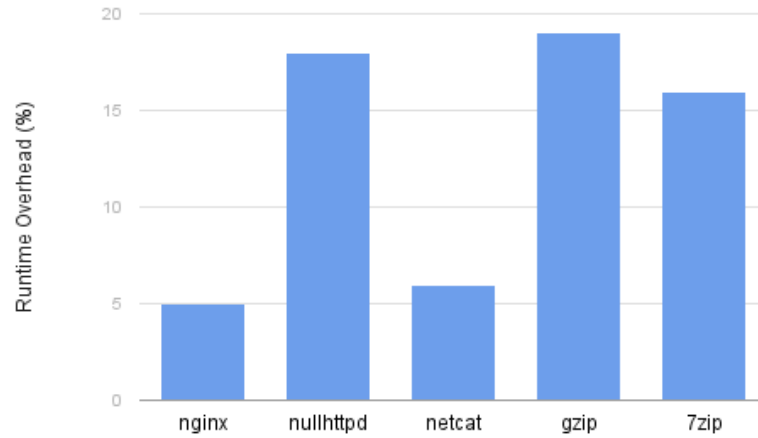


Figure 2.8: `sgx_malloc()` overhead in five non-trivial open source application.

to import critical data if the data has to be initialized by code outside SGX. Once critical data is imported into SGX, it will keep as secret from any code outside SGX.

cda_export: The critical data within SGX usually should not be exported to an outside SGX. For certain cases, export APIs can be used to export critical data if the import parameter indicates the data is exportable later. Otherwise, the API just fails.

cda_create: The Creation API is used to create a dedicated size SGX buffer for critical data, which can be read or written later. This can indicate the read/write permission for late usage.

cda_destroy: The Destroy API is used to destroy the SGX buffer which keeps SGX memory for critical data.

cda_read: The Read API provides the functions to read critical data with SGX.

cda_write: The Write API provides function to write critical data with SGX.

cda_encrypt: The Encrypt API provides some encryption capability once the key is imported into SGX as critical data. The application can use this API to encrypt data with the imported key.

cda_decrypt: The Decrypt API provide some decryption capability once the key is imported into SGX as critical data. Application can use this API to decrypt data with imported key.

7 Evaluation

In this section, we will demonstrate the utility of `sgx_malloc()` in protecting meta-data from exposure in user program memory. We evaluate the performance overhead

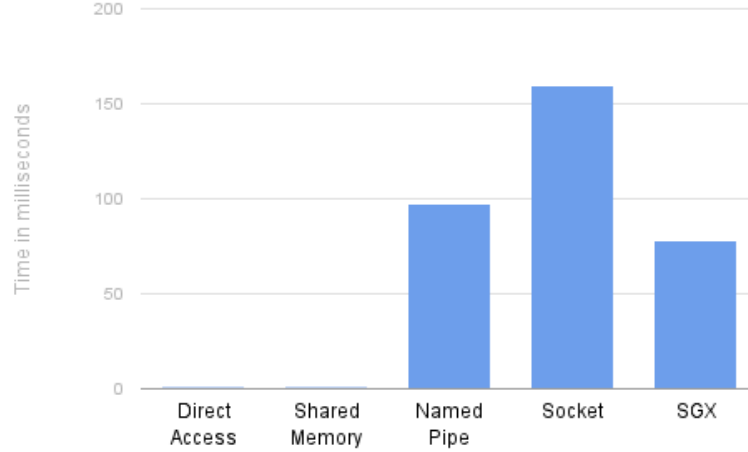


Figure 2.9: Local memory access, commonly used IPC methods and SGX’s performance, each access is repeated 10000 times.

of our memory allocator. We also demonstrate the utility of our SGX CDA API. We provided a sample use case in which the API was used to protect the private key of the corresponding web server certificate in openssl, which was the first priority target of the Heartbleed exploit[46]. The experiments were performed on a 64-bit Windows 10 Operating System running on a 6th generation Intel CPU with SGX enabled. The Intel SGX SDK was also installed on the operating system.

Overall Performance. We chose five non-trivial open-source applications to demonstrate the practicality of our solution. Each of the applications was modified to use our memory allocator APIs. Most importantly, our solution can hide the heap metadata with a guaranteed hardware-separated memory mechanism, reducing the attack surface by design. We use a program to show that even though the metadata’s address is known to the attacker, it still cannot be corrupted. We also apply our solution to previously mentioned software whose glibc library is vulnerable to the aforementioned GHOST exploit. It is obvious that once the memory allocator is changed, the attack mostly likely won’t work. However, we show that there is no attack surface for the metadata. Although an attacker can still exploit heap overflow vulnerabilities, e.g., by overwriting adjacent application data structures, the attacker cannot leverage the metadata for exploitation purposes.

Figure 2.8 shows the performance overhead of `sgx_malloc()` in the five open

source applications. The first two open source applications are widely used web servers: Nginx and `nullhttpd`. We test their performance by counting the response time per request after modifying the applications with our APIs. The overhead is low due to the web servers only needing a few `malloc` calls to complete one web request. We also measure the performance overhead in our `sgx_malloc()` solution. We run a pair of `sgx_malloc()` and `sgx_free()` calls 10000 times to accumulate time. As a comparison, we take the inner part of allocator out of enclave and convert it into a set of normal functions. Although our heap allocator algorithm is relatively easy, the process causes about a 400 percent overhead. The overhead is mostly due to the cost of SGX runtime library calls and environment switches caused by the SGX instructions. Although the performance overhead is relatively high, the solution provides a much stronger security guarantee. For the purpose of testing, and because the heap allocator is considered a public resource for the whole system, we use a debug version of the enclave which supports software debugging and use a test private key stored in the build system to sign the generated enclave file. A certificate may not be needed for the release version of enclave code as operating systems can police the launching of certain enclaves. This could also improve the performance overhead in further.

Another alternative isolation technique is to place the critical data into another process space. Programs can make an inter-process communication call to the other process asking for data. We list several common methods to exchange data between processes and measure their performance.

Figure 2.9 shows the performance for several commonly used inter-process communication methods. The test involved simple accesses of a variable in another process. To accumulate the result, each access is repeated 10000 times. Direct access to the memory address space costs less than 1 millisecond of performance time. Shared memory accesses, which is the fastest inter-process communication mechanism, also cost less than 1 millisecond. The operating system maps memory pages in the memory address space of two or more processes. Underneath different processes access the same physical pages. Its performance is the same as direct memory access within a process. However,

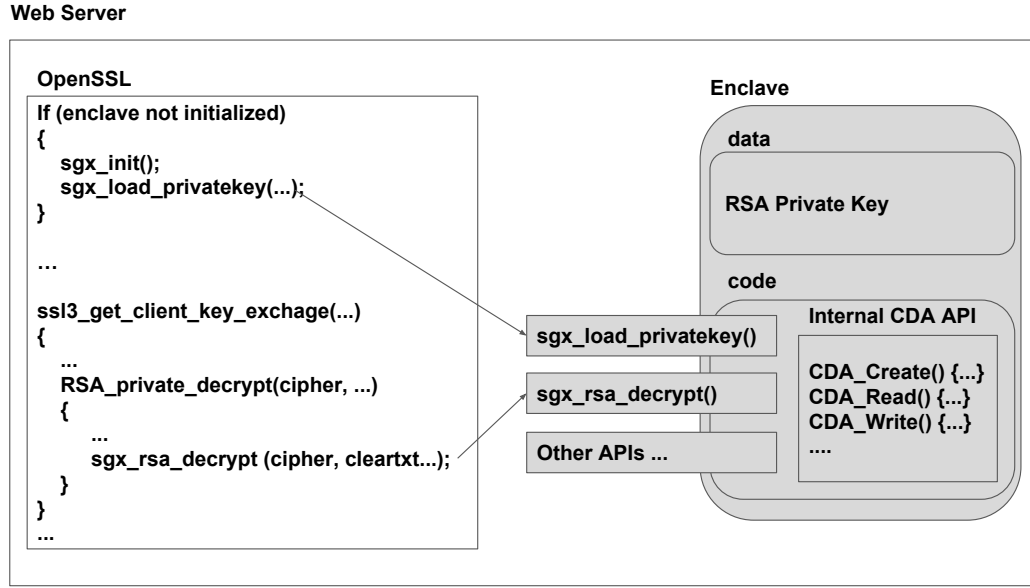


Figure 2.10: METASAFE implementation of OpenSSL with SGX enclave. The RSA private key is stored in the enclave, while METASAFE provides an interface to the internal CDA APIs that facilitates typical functions with the RSA private key, such as decryption. This mechanism would protect the RSA public key in spite of zero-day exploits such as the Heartbleed vulnerability.

it does not provide any security guarantee such as isolation. Socket and named/un-named pipe communication are commonly used for inter-process communication. Our results show that SGX performs better.

Case Study: Heartbleed Defense. The HeartBleed is a serious vulnerability in the OpenSSL library. The vulnerability allows the stealing of critical information that is normally protected by SSL/TLS encryption. The vulnerability is due to a bug in the `tls1_process_heartbeat()` and `dtls1_process_heartbeat()` functions. In a nutshell, an attacker can specify an arbitrary memory size for a message request regardless of the actual size of the message. Therefore, an attacker can return a buffer much larger than the expected message, leading to possible information leaks.

An attacker can read up to 65535 bytes of the server's memory. The memory may contain confidential information such as user names and passwords of web applications or it could leak memory layout information for a future attack that wants to bypass ASLR protection mechanisms. Even more critically, the private key of the web server's certificate can be exposed. With the private key in hand, an attacker can pretend to

be the legit web server using HTTPS. Once this happens, the certificate may not be recoverable. The owner of the server needs to revoke the certificate and redistribute a new one. Therefore, we implemented the OpenSSL library with METASAFE in order to provide an example as to how critical data such as the private key can be protected from a vulnerability like Heartbleed.

Figure 2.10 depicts our implementation of part of the OpenSSL SSL3 protocol. This protection was implemented by first loading the encrypted private key in PEM format into the enclave. When required, the key is decrypted inside of the enclave. The password for the PEM file is later sent into the enclave. After use, it will immediately be erased from memory.

The METASAFE implementation of this protocol required modification of a few OpenSSL functions. Because the RSA private key is now stored in the enclave, essentially any function that directly accesses the RSA private key needs to be modified to call the associated METASAFE API function call. For example, the `RSA_private_decrypt()` function is used to decrypt the RSA cipher. We modified the `RSA_private_decrypt()` function to call the METASAFE function wrapper `sgx_rsa_decrypt()`, which passes the cipher's address into the enclave and lets the built-in functions inside the enclave copy the value of the cipher. A buffer is also provided to receive the clear text. Within the enclave, we did not implement our own version of the RSA algorithm. We ported a version from the mbedTLS library. Because the functions are inside the enclave, the functionality of several of the APIs are rendered useless. For example, any APIs that involve I/O operations, calls to system services, or references to Win32 APIs cannot be implemented within the enclave directly. Only a limited C runtime library is available for use. To make it more straight forward and safe, any outside enclave system API is not used.

Performance Overhead and Exploitability. Because the RSA key exchange algorithm is only used once per session, the performance overhead introduced by METASAFE implementation is almost 0. The only overhead introduced are the additional instructions used to call the functions of METASAFE's OpenSSL function implementations.

Additionally, we evaluated our modified version of OpenSSL against the aforementioned Heartbleed exploit. We used an implemented Metasploit module that is used to exploit programs with the Heartbleed vulnerability[119]. Furthermore, we used WinDbg [142] to observe the memory manually to see if the private key was exposed. In both cases, the private key was securely stored in the SGX enclave and could not be accessed by either method.

Case Study: IE Browser Simulation. Since we cannot test our solution against the arbitrary memory writing vulnerability in the IE browser, a custom program is made to simulate the same environment. As Figure 2.7 depicted, `g_security_settings` is the global variable that needs to be protected. Assuming the attacker has the capability of arbitrary memory writing, if `g_security_settings` is changed anywhere besides the function `set_security()` using `sgx_store_data()` and the value that function `func()` receives is equal to 2, then the attacker is considered to have escalated privilege.

We assume that the program first calls `set_security()`, which verifies the token and sets `g_security_settings` to 0. In between `set_security()` and `func()`, the attacker finds the address of `g_security_settings` in memory and uses an arbitrary memory writing vulnerability to set it to 2 in order to escalate privilege. Later, `func()` is called to check the program's privilege and perform correspondingly.

```
sgx_store_data(&g_security_settings, 0);
...
g_security_settings = 2;
...
func();
```

The results show that even though `g_security_settings` is set to 2, `func()` still gets the correct value of 0.

The global variable `g_security_settings` that is defined in the program is just a placeholder. It provides an index into the private data inside of the enclave. Since the enclave memory space is protected by hardware, an attacker cannot access them unless there is a call to the corresponding functions. We also tested the case in which an attacker gets the address of the private data that represents `g_security_settings`

in enclave memory and tried to access it directly. This resulted in a memory access violation being raised.

There is still a footprint left in the program’s memory. The temporary variable `tmp` will hold the real value of `g_security_settings` for a limited amount of time. This means that once the real value is transferred from the enclave to a temporary variable that exists on the stack, it will still be vulnerable to buffer overflows within the function. This is demonstrated in the following assembly code representation of the function `func()`.

```
...
    tmp = sgx_get_data(&g_security_settings);
011D161E  push      11D9130h
011D1623  call      _sgx_get_data (011D10E6h)
011D1628  add       esp,4
011D162B  mov       dword ptr [tmp],eax

    if (2 == tmp)
011D162E  cmp       dword ptr [tmp],2
011D1632  jne       func+4Dh (011D164Dh)
...
```

We argue that the attack surface is reduced significantly. Because the address of the local variable exists on stack as a variable, it depends on the call stack. Once the function returns, local variables will be invalid even when there is a reentry to the function. Furthermore, ASLR makes it more impractical for the address of the local variable to be a target for arbitrary memory writing. We also argue that the local variable should not hold the value for too long before use, while value should be retrieved every time.

8 Related Work

In this section we investigate the state-of-the-art implementations of the heap. To address those weaknesses in various heap implementations, many mitigation methods focusing on protecting heap metadata have been introduced into the representative

memory allocators in the past decades. We analyze several popular mitigation approaches and notice that, as a result of integrating effective protection mechanisms, it has become less likely nowadays for an adversary to overwrite heap metadata (e.g., a doubly link list implementation) directly to achieve the capability of arbitrary memory writing. However, the adversary may still be able to overwrite heap metadata to steal the control of users' internal data, such as objects and settings, and hence exploit a further advantage.

Dlmalloc and ptmalloc. *Dlmalloc* is one of the most widely used memory allocators. Another well-known memory allocator *ptmalloc*, which is based on *Dlmalloc*, is the default memory allocator for GNU libc in Linux systems. One important feature of *Dlmalloc* and its derivatives, such as *ptmalloc*, is the inband metadata, which consists of chunk size and usage flags. An unallocated chunk uses a doubly linked list pointing to other available chunks. The use of the doubly linked list data structure is convenient for legitimate users. However, it also facilitates attackers as once there is a buffer overflow that overwrites the doubly linked list, attackers can easily access arbitrary memory space. To address this issue, a link pointer checking mechanism is introduced later. Unfortunately, this improvement is not sufficient enough, and there are still other ways to leverage a heap buffer overflow. Take the GHOST vulnerability which was discovered in 2014 [130]. The adversary overwrites the size field of the next contiguous free buffer to a larger size than it should be. This leads to overlapping the internal data structure of the user software, which is the Exim's internal memory allocator. This leads to an overlap of the internal data structure of the user software, known as Exim's internal memory allocator, and makes the internal data structure subject to malicious modification, such as arbitrary memory writing or information leakage.

Windows heap. Many vulnerability mitigation techniques have been implemented for the Windows heap since Windows XP SP2. They generally fall into two categories: metadata protection and non-determinism [155] [159] [171]. Because the majority of public exploitation techniques have traditionally relied on the corruption of one or more heap data structures, metadata protection is their focus. Windows heap uses free lists, which are several doubly linked lists for different memory chunk sizes ranging from

16 bytes to 1024 bytes[116]. Although the free list data structure is fast, it relies upon metadata, making the structure a target for attackers. The technique to exploit a doubly linked list is the same as used in the dlmalloc exploit, in which an attacker uses the unlink operation performed during the coalescing of three chunks to achieve the capability of arbitrary memory writing. Since Windows XP SP2, integrity checking has been implemented to verify the process of unlinking any list. The integrity of the structure member `Entry->Flink->Blink` is verified. With our jemalloc design, we already prevent similar attacks by design.

Several mitigation techniques have also been applied to protect the metadata in the heap manager from corruption, such as the use of a heap entry header cookie, heap entry metadata randomization, and function pointer encoding. For example, an 8-bit random value was added to the header of each heap entry to be validated when a heap entry is freed. In Windows Vista, the cookie is extended into a random 32-bit value which is XORed with each heap entry. The heap manager then unpacks and verifies the integrity of each heap entry prior to operating on it[103][152][154]. It's become very difficult for the attackers to abuse the heap metadata without crashing the program.

Furthermore, Guard Pages have been used for heap management since Windows 8. They are added between `_HEAP_USERDATA_HEADER` objects. Therefore, an overflow will need to exist in the same UserBlock in order to be effective. However, one guard page is very costly, using 4KB of memory (one page) just to act as a tripwire. To reduce the overhead, the Windows 8 heap manager decides to add guard pages for the subsequent UserBlock only when one user block is big enough. This allows attackers to avoid the trigger guard page by finding an unprotected portion of the metadata[179]. In Windows 8, an attacker can manage to overwrite `HEAP_USERDATA_HEADER`'s `FirstAllocationOffset`. This allows the attacker to control the allocation of new properly-sized objects.

From the evolution of Windows heap manager we can see that the reason for most vulnerabilities is not because the heap manager is poorly implemented. The vulnerabilities arise when programs use the heap buffer improperly so that they are left susceptible to buffer overflows. When a buffer overflow happens, metadata is no different than any

other part of memory and hence is vulnerable to any overwriting. In our design, we propose that the metadata be placed inside of an enclave. The chances of the heap manager internal functions inside the enclave having vulnerabilities is very low. If bugs are found, they can be fixed without changing any of the data structures.

DieHarder. Dieharder[124] stores metadata separately from user buffers. To prevent metadata from being overwritten, it also uses a guard page for isolation. This solution could be vulnerable to arbitrary memory writing vulnerability, even though with such an ability an attacker may choose a more valued target such as critical global settings.

9 Conclusions

In this paper, we presented METASAFE, a solution that leverages Intel’s Software Guard Extensions (SGX) to provide hardware-separated memory space using enclaves to protect critical data, especially the metadata of certain APIs. In particular, we implemented a novel set of heap management APIs that leverage Intel’s SGX to protect against tampering of heap metadata by securing the critical data within SGX’s enclaves. METASAFE protects critical data from buffer overflow attacks, including the novel set of buffer overflow attacks on non-control data such as DOP attacks. Additionally, we introduced a suite designed for ease of use so that the programmers can protect and manage critical data in the user programs without significantly modifying the current programming model. We evaluated the practicality and overhead of METASAFE on several open source projects and found the results to be very promising. Furthermore, we implemented a use case that shows how METASAFE can be used to protect against the exposure of critical data of user programs via zero-day exploits such as the Heartbleed vulnerability.

Chapter 3

Use-After-Free Mitigation via Protected Heap Allocation

1 Introduction

Despite ongoing battle against software attacks, memory corruption vulnerabilities are still discovered in popular applications and exploited to provide adversaries with capabilities such as remote arbitrary code execution. Specifically, use-after-free (UAF) vulnerability, as a class of memory corruption bugs, play a dominant role in exploiting complex software packages. UAFs are prevalent in latest releases of popular browsers such as Tor, Internet Explorer, Chrome, Safari, and Firefox [115].

According to the Common Vulnerabilities and Exposures (CVE) statistics, almost one in every five reported arbitrary code execution vulnerabilities in 2016 originated from UAF bugs [110], where a pointer to a freed object is dereferenced by mistake. More specifically on Firefox, out of the 420 CVE vulnerabilities between 2014-2016, 194 (46%) were UAF vulnerabilities [41]. In Pwn2Own 2014 [63], an annual contest among hackers and security research groups, the VUPEN team was awarded with the largest cash amount, \$100,000, for a single UAF exploit that affects all major WebKit-based browsers. A common exploitable feature of the aforementioned software is their client-side scripting support (e.g., JavaScript and ActionScript). This enables adversaries to invoke memory allocation and rearrange memory layout in order to complete the attack by sending in the malicious payload, e.g., return-oriented programming (ROP) chain [136].

Despite deployed mitigations by various software vendors (e.g., Mozilla, Apple, Google, and Microsoft) and research groups, defense against UAF exploits remains unsolved. Noteworthy attempts to address UAF include heap canaries [135], memory

isolation (e.g., isolated heap (IH) [117]), deferred freeing (protected free [59]), pointer-seeking garbage collection (e.g., MemGC [172]), and random heap allocation schemes (e.g., DieHarder [124] and Windows 8 low fragmentation heap (LFH) [158]). As a recent mechanism in real world, Microsoft IH [101] allocates different memory regions for different “types” of objects. This makes it difficult for the adversaries to replace a freed object content in the isolated heap with controlled malicious data. As the result, IH assumes all pointer dereferences later are safe. Although IH raised the difficulty for UAF attacks, it was later broken [2].

In this paper, we present ZEUS that protects web browsers against UAF exploitations using fine-grained randomness. Although coarse-grained randomization plays a major role in memory security nowadays, it still cannot completely protect against fine-grained attacks such as UAF in web browsers [117]. We show that fine-grained object address perturbation can prevent certain UAF exploits in web browsers. ZEUS makes the memory allocation outcomes locally unpredictable. Hence, even in the presence of UAF vulnerabilities, the adversaries cannot precisely align past (freed) and currently allocated object images and the corresponding in-object member fields.

Upon each allocation, ZEUS perturbs the object addresses randomly given an extra memory buffer within the same allocation region. The introduced random prefix offset at the beginning of each allocated object prevents the adversaries from refilling the exact target freed memory addresses with malicious content. Consequently, the follow-up dereferencing of the misaligned malicious content results in a process crash due to segmentation faults. ZEUS also provides protection against later stages of UAF exploitations. In a typical UAF scenario, the adversary launches a code reuse (e.g., return-oriented programming - ROP) attack after refilling a recently freed memory. The payload is previously prepared in memory using heap spray mechanisms [124]. The first gadget in code reuse is often stack pivot [43] to switch stacks. ZEUS’s nondeterminism provides an extra layer of defense against precise determination of the stack pivot address in sprayed heap regions.

Compared to existing UAF mitigations, ZEUS provides a application-agnostic alternative, i.e., it provides protection for closed-source software binaries without the need

for tedious discovery of individual UAF vulnerable points. ZEUS’s unpredictable allocation outcomes prevents UAF exploits that require exact object alignment and often rely on deterministic memory layout. Incorporation of randomness into memory allocation have been introduced in the past work [124]. These include fundamentally different solutions such as randomization of the heap’s base address [18], random heap chunk sizes [74], random chunk selection [158], and random heap meta-data location [124]. All the past work that use non-determinism randomly modifies the location and size of each heap object to be aligned at the multiples of its size. This leaves them vulnerable to heap spray attacks that make sure one of the sprayed locations aligns with the target freed object. ZEUS deploys fine-grained non-determinism within each region. It introduces prefix random offsets to individual objects independently, and violates in-object member field alignment.

We have implemented and validated ZEUS against several CVE vulnerability exploitations in large and popular software packages such as Firefox and Tor web browsers. ZEUS terminates the exploits at early stages of the attack in all those cases with negligible 1.2% runtime performance overhead. ZEUS requires additional dynamic memory space because of introducing random allocation offsets. The overall extra process memory requirement can be adjusted to optimize the trade-off between lowering the success probability of an exploit and smaller additional random offsets. Based on our experiments, lowering the probability of an attack against Firefox’s CVE-2016-9079 vulnerability to 0.0069 leads to 15% of overall extra memory space. Given ZEUS’s very low runtime performance overhead, we believe it is suitable for most practical settings, where dynamic performance overhead is penalized much more heavily compared to extra memory requirements.

The contributions of this are as follows:

- We propose ZEUS, a secure practical memory allocation design that prevents UAF vulnerability exploitations by using fine-grained randomness.
- We implemented ZEUS and applied it to well known complex and large scale software packages (Firefox and Tor web browsers).

- We validate ZEUS against several CVE exploits. Our results show, with negligible performance overhead (1.2% on average), ZEUS is resilient to sophisticated bypassing techniques.

Threat model. We focus on protecting web browsers against UAF exploitations. We assume the adversary has the power to launch remote attacks and allocate and free objects at will to exploit UAF vulnerabilities. As shown in practice, such attacks are most useful against software packages that combine client-side scripting languages with back-end operations such as web browsers, where the adversary has the unlimited ability to allocate and free objects through executing JavaScript or Flash code. This model thus assumes the worst-case for prime attack targets in the real world. Furthermore, unlike server applications, web browsers’ context does not facilitate repeated brute-force attacks for the adversary to repeatedly launch the attack if the previous attempts fail. In a web browser, if the first attempt fails and causes the browser to crash, the user may not attempt to reload the page. The attack has often only one chance to succeed per target. Therefore, ZEUS does not assume repeated attacks as its threat model.

2 ZEUS’s Design

ZEUS’s objective is to prevent UAF exploitations in real software systems, where there may be no knowledge about the source code details or access to the source code at all. The recent emerging popularity of UAF exploitation among malware in the wild, their success despite the existing proposed defense mechanisms, and complications of discovering UAF bugs in large software packages further motivates the need for a practically deployable protection scheme.

The core reason that UAF attacks succeed is the adversarial knowledge and possible control over the exact fine-grained heap memory data layout. Predictable layout leaves many of the existing mitigations ineffective. For instance, a recent exploit [2] leveraged the fine-grained determinism in memory layout to complete the UAF steps for remote code execution despite Microsoft’s state-of-the-art IH defense.

ZEUS redesigns heap allocation functionality such that every allocation leads to

a proceeding offset of a random length before each memory object. For backward legacy compatibility and to ensure the randomness will not compromise the heap layout maneuver (so-called heap massage [151]), ZEUS uses and expands the concept of memory *regions* in popular heap allocators such as Jemalloc, used in known widely-used software such as Firefox and Tor web browsers. The added random prefix makes subsequent heap allocation outcomes (e.g., in-object member field addresses) unpredictable. Hence, the random prefixes prevent malicious deterministic planning, layout control, and exploitation to complete UAF steps.

Traditional heap allocators use free lists to maintain runtime information about available memory areas. Depending on the dynamic allocation trace, a free list node may split after an allocation, and two adjacent nodes may merge together following a free operation to lower memory fragmentation. UAF exploits use such a design to take control of the heap layout evolution over the program execution. For instance, a carefully crafted JavaScript code can make the web browser reallocate the memory area for a recently freed large object with several smaller objects, or vice versa.

Inspired by seminal research results [17], state-of-the-art heap allocators segregate meta-data from user data, and partition the memory space into a hierarchy of fixed-sized buffers, so-called *chunks*, *runs*, and *regions*. Regions are the basic heap items that reside in predetermined memory addresses, and are returned to the program upon allocation calls. Unlike traditional free list nodes, regions neither split nor merge together. Therefore, for a UAF exploit to reallocate a recently freed object memory, the new object should be almost the same size of the original one. Otherwise, a different run and region with a suitable size will be picked by the allocator. Fixed-size region-based allocation schemes further raise the security bar against UAF exploits by limiting their options. However, several attacks have been reported against them because of predictable region addresses [133].

Recent research work [124] proposed coarse-grained non-determinism through random selection of regions for object allocations. However, location of objects at the beginning of the randomly selected regions by the allocators leave them susceptible to heap spray attacks. Those attacks exploit the fine-grained determinism in allocators to

align one of the sprayed objects in the target region that previously hosted the recently-freed object of interest. ZEUS protects against those attacks by providing fine-grained non-determinism.

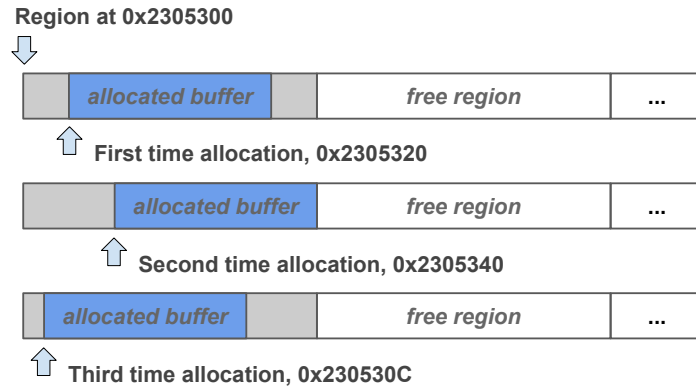


Figure 3.1: Region-based allocations with random prefix.

For each memory request by the program, ZEUS allocates an area that is larger than the requested amount by a randomly selected margin. the margin is selected for individual memory requests independently. ZEUS uses the introduced extra memory buffer as the prefix to the object location within the region. Consequently, the objects are not anymore located at the beginning of the regions, and their exact location within the region is not predictable to adversaries. Figure 3.1 shows a sample outcome, where individual allocation calls are assigned a random number as the prefix offset to the allocated memory. Hence, malicious same-size memory allocation after an object free operation will result in misalignment between the original and new objects, and hence their fields such as the target virtual table pointer used for UAF exploits. As the result, the maliciously refilled target pointer value will not match the adversary’s objective value. Hence, the follow-up pointer dereferencing does not reference the adversary’s buffer and the attack fails.

In case of a “lucky” alignment between the new and the original freed objects, ZEUS’s extra lines of defense prevent the next steps of the attack by randomizing the allocated heap buffer that contains the stack pivot gadget address. Figure 3.2 shows how ZEUS allocator causes a misalignment among the set of heap-sprayed ROP payloads. Each illustrated payload includes a padding, stack pivot address, return chain, and

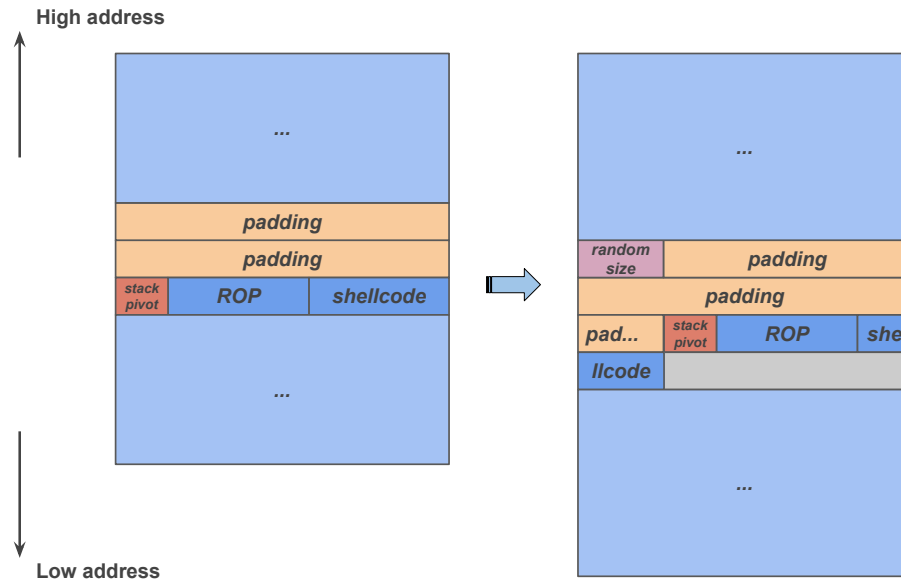


Figure 3.2: ROP chain after unpredictable heap spray.

possibly a shell code if ROP deactivates DEP. Consequently, the exploit cannot make the instruction pointer register (eip) point to the stack pivot address deterministically.

2.1 Optimization

We discuss ZEUS's dynamic system resource consumption. Its built-in protection does not require any binary or source code instrumentation, or any explicit runtime checks. These are among the reasons for ZEUS negligible runtime performance overhead. On the other hand, its adjustable extra use of memory because of slightly enlarged memory allocations plays as a trade-off vs. improved protection, i.e., larger object prefixes result in less lucky adversaries. We have taken steps to optimize ZEUS's design to reduce its memory overhead on web browser's runtime execution.

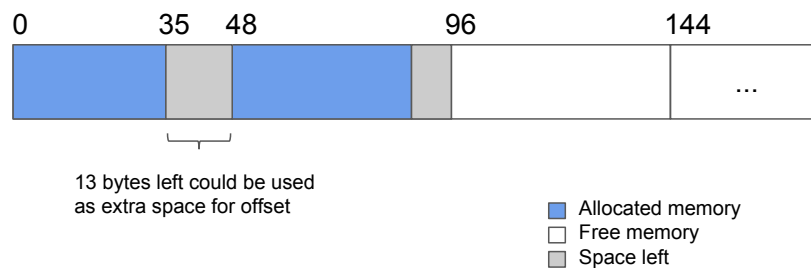


Figure 3.3: Wasted memory used for random allocation.

The allocator maintains fixed sized memory regions of various lengths from 2 bytes to regions of 2048 bytes each or even one whole page. Upon a memory request, the heap allocator designates the smallest memory region that is bigger than the requested buffer size. The difference between the requested amount r and the selected region size s often leaves a wasted buffer (with size $0 \leq s - r$) within the region that is not used for the user data (Figure 3.3). If the wasted buffer’s size exceeds ZEUS’s predefined minimum threshold τ (i.e., $s - r \leq \tau$), ZEUS leverages the space as its “leeway” to randomize the allocated memory’s address within the region. ZEUS picks a random number p uniformly from $[0, \tau]$ as the prefix size for the object within the region. Consequently, ZEUS’s protection does not introduce runtime memory overhead for the aforementioned request types, where the requested size plus the threshold fits in the region (i.e., $r + \tau \leq s$). Otherwise, ZEUS allocates the requested buffer in the smallest free region with the size larger than the request buffer plus the threshold value.

As an optimization to take full advantage of the provided entropy source by the allocator, ZEUS’s implementation picks the random number p within anywhere in the available wasted buffer, i.e., $p \in [0, s - r]$. Based on our experiments with Firefox, with a threshold of 48 bytes, approximately 29% of memory allocations on average fall into the first category above, and do not require extra memory space when ZEUS is deployed.

It is noteworthy that ZEUS’s current implementation does not consider any application specific features. However, if such information is available, ZEUS can take advantage potentially. For instance, not all memory allocations in a program may require ZEUS’s protection. As a case in point, the data structures and buffers that do not include UAF-sensitive fields such as pointers may be exempted; ZEUS can switch to the original deterministic allocator when a white-listed buffer allocation is invoked by the program.

2.2 Security

When choosing where to allocate a new object, ZEUS chooses a fine-grained prefix offset p uniformly between zero and the predefined threshold τ , so that the object

is not located at the beginning of the memory region *deterministically*. It provides sufficient and adjustable level of entropy against probabilistic UAF attacks that target random offsets by making a random guess about the offsets. It is noteworthy that the allocated object addresses can be disclosed through information leaking vulnerabilities that leave not only ZEUS but also many past mitigation mechanisms (e.g., ASLR) ineffective. ZEUS provides partial defense against information leaking exploits by randomizing individual object allocations independently; therefore, launching a UAF attack requires deployment of information leaking attacks to disclose the reallocated object address and stack pivot address in heap memory.

We analyze the entropy level provided by ZEUS, and calculate how the probability of success for the attack varies given a allocation offset threshold value τ . Targeting a single allocation succeeds with probability of $\frac{1}{\tau}$. Hence the success rate of the UAF's first step (alignment and refilling the freed object content, e.g., virtual table pointer value) is $\frac{1}{\tau}$. The second step (dereferencing a member function that requires two indirections through virtual function table) will succeed with probability of $\frac{1}{\tau^2}$ that leads to execution of the stack pivot. Similarly, modification a certain field of a string object, as the attacker tries to obtain the read primitive to bypass ASLR, will have a success probability of $\frac{1}{\tau}$. Consequently, the whole UAF attack succeeds with probability of $\frac{1}{\tau^3}$ because of the independence among ZEUS's choices of random numbers for individual allocations. As a concrete example, with an offset threshold size of 16 bytes, ZEUS can pick p to be any element in the following set: $A = \{0, 4, 8, 12, 16\}$ ¹. Hence, $\tau \leftarrow |A| = 5$, and the attack success rate drops to $\frac{1}{5^3} = 0.008$. Increasing the offset to 48 bytes diminishes the success rate significantly down to 0.00045.

In our calculations above, we only considered the minimum number of steps that are required for a successful UAF attack. However, successful completion of a UAF attack in real world requires several other steps that rely on deterministic heap allocation outcomes. We review details of real CVE exploits in Section 4.4. Figure 3.10 shows a sample inter-memory dependencies that need deterministic layout for the attack to

¹Selection of numbers that not coefficient of 4 is not possible in most of the programs due to the x86 memory alignment requirements.

succeed. We did not consider these steps in our calculations conservatively.

3 Implementation

We implemented ZEUS and used it in real-world popular web browsers.

Interface To ensure legacy compliance, ZEUS provides a typical heap allocation interface to web browsers, and keeps its protection mechanisms transparent to the application. Traditional C dynamic allocation functions such as `malloc`, `free`, `calloc`, `realloc` are supported. Furthermore, ZEUS provides extra interfacing API such as additional default-valued arguments to the aforementioned functions so that the developers can optimize the overhead-vs.-security trade-off. The extra API functionalities may be ignored by the program, and in that case, the default settings will be applied. As the result, ZEUS works with closed-source programs out-of-the-box without any changes required.

Randomization granularity If a browser does not require typical x86 memory alignment (e.g., 4 bytes in 32-bit systems for improved data cache performance), ZEUS’s implementation increases the randomness granularity of memory allocations to 1 byte. This would lead to a larger entropy source. Therefore, one can decrease the minimum required randomization range (threshold τ) without affecting the scheme’s overall security. We evaluate the performance of different randomization entropy levels in the next section.

Randomization offset According to our observations (see, for instance, Figure 3.4 for Firefox), most of the memory allocations by popular browsers are for medium-sized buffers (from 16 to 1024 bytes). The recent UAF CVE exploits mostly target such medium-sized buffers, and even more so the larger buffers. This is because very small regions (less than 9 bytes in size) are infeasible candidates for heap sprays. Additionally, by design and unlike traditional free list-based solutions, the memory regions are not split or merged in ZEUS. Hence, our implementations do not randomize memory

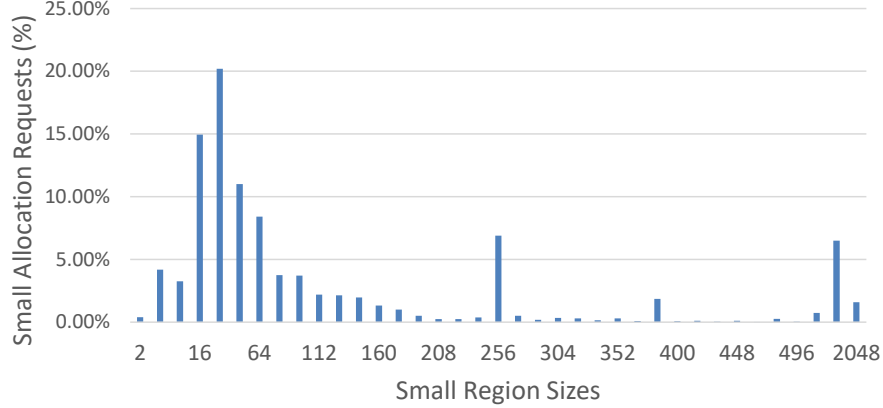


Figure 3.4: Firefox heap allocation request size frequency.

allocations for buffers with fewer than 9 bytes. On the other end of the spectrum, large buffers ($> 1KB$) are often used by real UAF exploits to spray the stack pivot’s address on the heap memory. Large allocations in real programs lead to larger wasted memory (discussed in the previous section). This provides ZEUS with a larger entropy source for its randomized allocations. As the result, more popular buffer sizes among UAF exploits are protected better by our implementations.

Reallocating The program may call `realloc` to increase or reduce the size of a previously allocated memory buffer on the heap. Traditionally, free-list allocators could resize and return the same pointer as passed in as the argument. Like existing state-of-the-art allocators, ZEUS does not coalesce adjacent regions to enlarge an allocated buffer. However, if the increased requested buffer size plus ZEUS’s randomization offset threshold fit in the same memory region, ZEUS will re-randomize and return a pointer to the new buffer within the same region. Otherwise, ZEUS treats the memory reallocation request as a brand new one. It copies the original buffer’s content to a new allocated one, and releases the original buffer memory.

4 Evaluation

We evaluated ZEUS and validated its practicality when used in real-world popular web browsers (four different versions of Firefox and Tor) without source code modification requirements, and against several real CVE exploits. We performed our experiments

on Intel Core i5-3337U CPU with 8GB of RAM, and running x86_64 Windows 7 operating system. In particular, our experiments answered the following questions: *i)* what is ZEUS’s runtime performance and memory overhead? *ii)* how well does ZEUS increase the security of the existing web browsers? *iii)* how does ZEUS compare to state-of-the-art commercial mitigation mechanisms against UAF exploits? and *iv)* how can ZEUS stop real past CVE UAF attacks?

4.1 Performance

ZEUS’s protection does not require runtime checks such as memory value checks or control flow monitoring to ensure protection against UAF. Its main runtime functionality is to generate random offsets and designate the corresponding memory regions upon heap allocation calls by the browser.

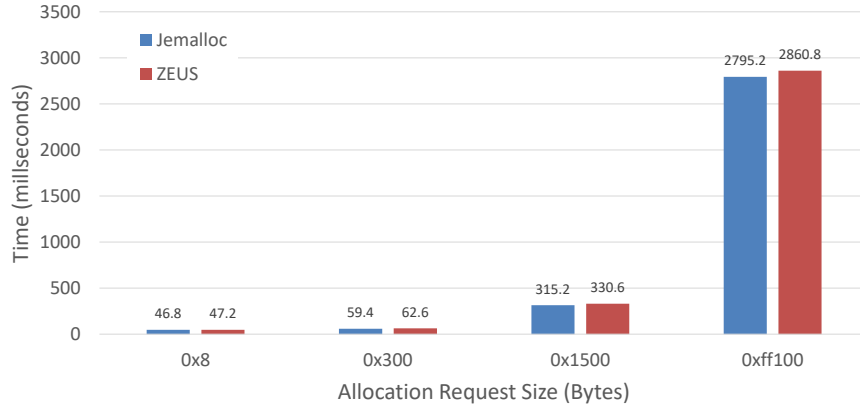


Figure 3.5: ZEUS’s performance overhead with different allocation request sizes.

Figure 3.5 shows our results for the runtime performance overhead caused by ZEUS’s protection in Firefox. We measured the time requirement for heap allocation for four different object sizes. The numbers are averaged over 10K runs. The results compare the ZEUS-protected vs. vanilla Firefox executions. The performance overhead is negligible, i.e., around 2.7% on average for individual allocation instances. ZEUS’s overall overhead on the application as a whole (Firefox) is lower, i.e., 1.2% on average.

We use Firefox’s network tool to monitor the loading time of different web pages. On each time, the loading time may varies significantly because of the dynamic part of the web page. Thus we choose to count the time for certain amount of requests for

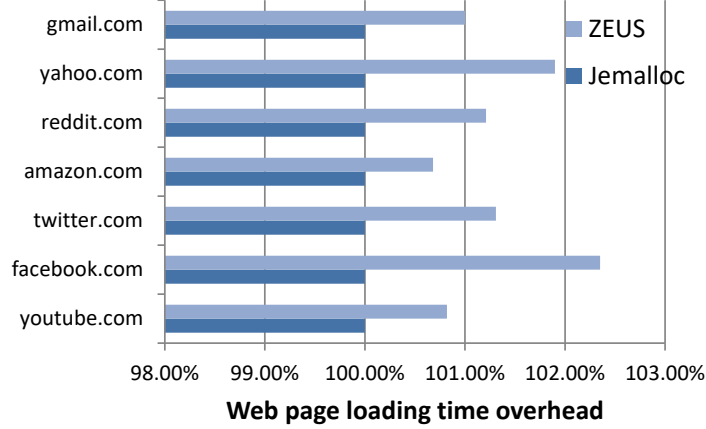


Figure 3.6: ZEUS’s performance overhead with loading different websites.

each page(each request for one element on the page).

We believe that ZEUS’s overhead is very promising and acceptable for real deployments in practice.

4.2 Memory

ZEUS’s introduced random offsets to allocated buffers cause runtime memory overhead to the browser.

Table 3.1: ZEUS’s overhead on browser’s memory (%).

Site URL	0x30 bytes	0x60 bytes	0x90 bytes
www.google.com	14.82	24.72	34.46
www.facebook.com	12.53	26.09	32.3
www.quora.com	17.17	35.76	51.67
www.youtube.com	18.4	33.20	45.7
www.gmail.com	9.9	13.9	20.4

Table 3.1 shows the results for Firefox web browser, when surfing five popular websites. Numbers indicate the overall extra memory that the browser requires because of ZEUS’s deployed protection. Each column represents different minimum prefix offset sizes (threshold τ) for allocation address randomization. Use of 0x30 bytes as the randomization offset size results in very low success rate (0.00003) for probabilistic attacks, and causes a reasonable overall memory overhead (14.4% on average). The memory requirement does not double when moving from offset sizes of 0x30 to 0x60 bytes, because ZEUS uses the wasted memory within regions (discussed earlier in the

paper).

4.3 Security

We performed a security analysis to calculate the success probability of a UAF exploit despite ZEUS’s mitigation.

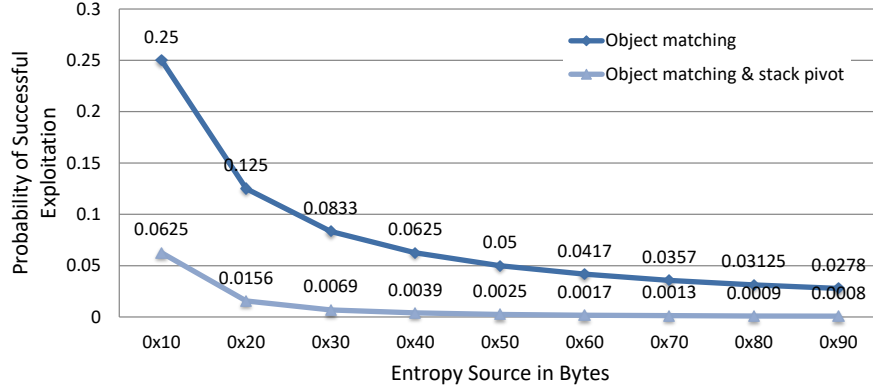


Figure 3.7: Exploit success probability against ZEUS.

Figure 3.7 shows the results, where the horizontal axis indicate the entropy source in bytes (randomization offset size). The vertical axis shows the success probability of a UAF exploit. The graph contains the rates for first step of the UAF exploitation (object matching) and the rates for the first two stages that are crucial to launch the stack pivot. In addition to the aforementioned analytical calculations, we also performed empirical tests. We ran the exploits 20K times, and counted the success rates. The results from empirical measurements matched those from the analytical calculations closely.

Table 3.2: Vulnerabilities used in the evaluation.

Program	Vulnerability	Result with ZEUS
Firefox 45.5.0	CVE-2016-9079	protected
Firefox 17.0	CVE-2013-0753	protected
Firefox 9.0.1	CVE-2011-3659	protected
Firefox 3.6.16	CVE-2011-0065	protected
Tor 6.4	CVE-2016-9079	protected

To validate ZEUS in real world, we evaluated ZEUS against five real CVE exploits (see Table 3.2 for the list). For exploits, we either implemented them ourselves given CVE information, or (if available) obtained them online or through Metasploit modules [111]. For instance, to exploit CVE-2011-3659 in `AttributeChildRemoved`

function of Firefox 9.0.1, we used Metasploit’s `browser/mozilla_attribchildremoved` module [134]. On default state-of-the-art heap allocators, the exploits were successful every time we ran them. Each session involved directing the browser to a malicious website that contained the crafted script.

Through its randomized allocations, ZEUS was able to terminate all the exploits at the early stages of UAF attacks before even the corresponding ROP chains launched. Using hardware read breakpoints in Windbg, we were able to inspect the randomized allocation offset values.

4.4 Case studies

We briefly explain ZEUS’s protection against real-world past UAF exploitations in Tor(Firefox) browsers.

CVE-2016-9079 The Tor web browser’s CVE-2016-9079 [114] was a recently discovered atypical and fairly complicated zero-day UAF exploit. It leverages an iterator pointer that could overpass the buffer limits.

The vulnerability resides the code snippet above, in function `NotifyTimeChange` [166]. The attacker puts `<svg>` JavaScript element, which uses `<animate>`, in the malicious web page. A `nsSMILTimeContainer` object is assigned to it to perform time bookkeeping for the animations. Each `nsSMILTimeContainer` has a member `mMilestoneEntries` array that organizes each event in the animation. Then a single call to `pauseAnimation` triggers the member function `NotifyTimeChange` shown in the snippet above. As illustrated, in the function, the pointer `p` iterates through the elements of the `mMilestoneEntries` array, and calls each element’s member function `HandleContainerTimeChange`. The call may potentially add elements to `mMilestoneEntries` increasing its size. If the size increase goes beyond the heap allocator’s memory region size, the array gets reallocated to a bigger region by the allocator automatically. This leaves the pointer `p` pointing to a freed memory space, and still iterating through the freed space by the while loop above beyond the boundaries of the original array.

```

1 void
2   nsSMILTimeContainer::NotifyTimeChange()
3 {
4   // Called when the container time changes w.r.t the document time.
5
6   const MilestoneEntry* p = mMilestoneEntries.Elements();
7   ...
8
9   while (p < mMilestoneEntries.Elements() + mMilestoneEntries.Length())
10    {
11      mozilla::dom::SVGAnimationElement* elem = p->mTimebase.get();
12      elem->TimedElement().HandleContainerTimeChange();
13      MOZ_ASSERT(queueLength == mMilestoneEntries.Length(),
14        "Call to HandleContainerTimeChange resulted in a change to the queue
15        of milestones");
16      ++p;
17    }
18 }

```

125e7100	00000000	00000000	00000000	00000000	300fff80	00000000	00000000	00000000
125e7120	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7140	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7160	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7180	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5
125e71a0	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5
125e71c0	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5
125e71e0	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5	e5e5e5e5
125e7200	00000000	00000000	00000000	00000000	300fff80	00000000	00000000	00000000
125e7220	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7240	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7260	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7280	00000000	00000000	00000000	00000000	300fff80	00000000	00000000	00000000
125e72a0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e72c0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e72e0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7300	00000000	00000000	00000000	00000000	300fff80	00000000	00000000	00000000
125e7320	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
125e7340	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 3.8: Tor’s memory snapshot after the heap sprays that surround the target buffer with crafted data.

ZEUS’s protection of CVE-2016-9079 is based on the fact that the attack exploits the determinism in controlling the memory layout. This atypical UAF exploit does not leave any time window after the free operation for the attacker to allocate the original

freed memory space, since all operations above happen during the `pauseAnimation` call. Hence, the attacker prepares the memory before the `pauseAnimation` call. Since `mMilestoneEntries`'s initial size is 0x80 bytes, the attacker starts by spraying 0x80 bytes of JavaScript ArrayBuffers and then sets up `animate` data to create the `mMilestoneEntries` array. Figure 3.8 shows our Tor browser's memory snapshot after the heap sprays. The crafted data surround the freed target `mMilestoneEntries`, which sits at 0x125e7180.

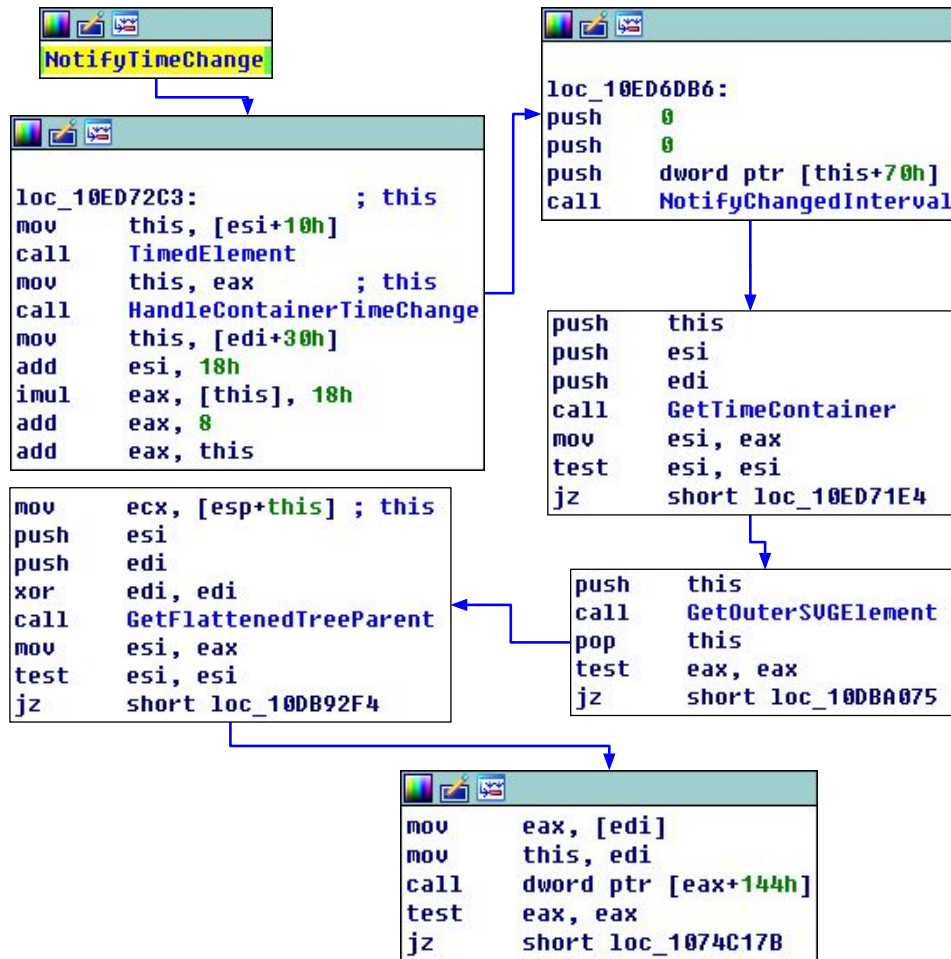


Figure 3.9: Call sequence for UAF attack in Tor browser.

In our experiments, once the freed pointer `p` went beyond the target buffer `mMilestoneEntries`, it landed on attacker-controlled sprayed data, from where the rest of the attack followed typical UAF exploit steps by hijacking the control flow. Figure 3.9 shows the function call sequence that led to an indirect call-site `call [eax+144h]`. The call-site was

later used for the control flow hijack by ultimately² copying the stack pivot address in memory address `eax+144h`.

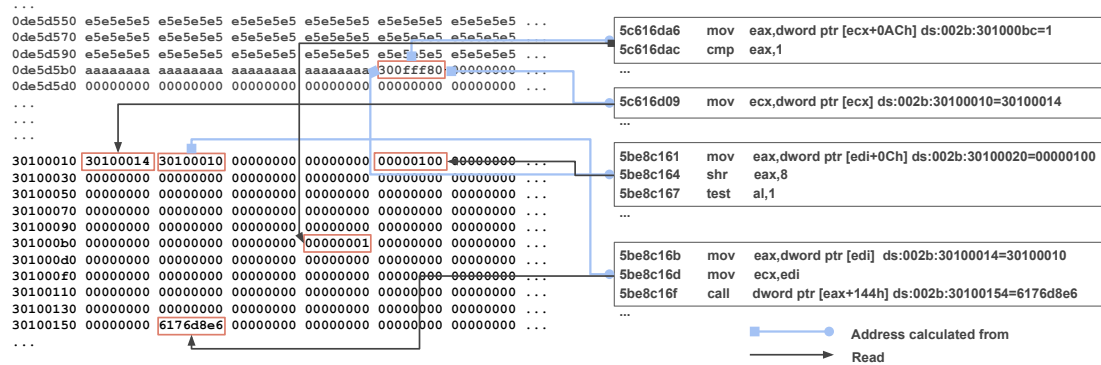


Figure 3.10: Exact manipulation of several addresses (pointer values) is required for CVE-2016-9079 attack success.

For ZEUS, it is noteworthy that the completion of the CVE-2016-9079 attack involves *i)* heap sprays at four different steps, and *ii)* several careful manipulation of pointer data to maliciously direct the control flow according to Figure 3.9. Figure 3.10 shows the memory snapshot that we took after the exploit succeeded. As illustrated, many pointer values need to be correctly updated to the exploit’s success ultimately. Consequently, determinism in heap allocation functionalities is the core weakness that CVE-2016-9079 exploits. ZEUS targets this weakness directly through its randomized allocations.

Once ZEUS’s mitigation was deployed, all allocated memory regions addresses were subject to a random change. With so many specific pointers that the exploit had to get right for the correct function of the malware, ZEUS’s randomized allocation lowered the possibility of a successful attack remarkably. The exploit resulted in a process crash all 10K times of our retrials. Figure 3.11 shows the memory snapshot during the attack with ZEUS enabled. The allocated areas with random addresses causes the intended malicious control flow to be not followed as planned. The region *right after* the prefix offset (shown in dashed box) is *supposed to be* under attacker’s control for the exploit to determine the exact subsequent addresses to replace with crafted values (300fff80). However, ZEUS’s randomized allocation removed that determinism in CVE-2016-9079

²We skip details here for space limitations.

```

0d7449f0 00000000 00000000 00000000 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5
0d744a10 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5
0d744a30 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5
0d744a50 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5
0d744a70 e5e5e5e5 e5e5e5e5 e5e5e5e5 e5e5e5e5 00000000 00000000 00000000 00000000
0d744a90 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744ab0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744ad0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744af0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744b10 00000000 00000000 00000000 300fff80 00000000 00000000 00000000 00000000
0d744b30 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744b50 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744b70 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744b90 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744bb0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744bd0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744bf0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744c10 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0d744c30 00000000 00000000 300fff80 00000000 00000000 00000000 00000000 00000000

```

Figure 3.11: Tor browser’s memory with ZEUS enabled.

(it reads 00000000 in the red box) leading to an exception and process crash even before the exploit got to the stack pivot launch point.

UAF exploits are often interested in large chunks of available memory space (e.g., 1-2 MB) to ensure deterministic control over the memory layout locally. When dealing with allocations larger than 1 MB, ZEUS puts its randomization buffer limit to one page (4096 bytes). This is relatively a small overall memory overhead, but it provides a strong protection by minimizing the possibility of a successful exploit that requires luck in all consequent steps of the attack as shown above in CVE-2016-9079. Figure 3.12 shows the probability of the exploit’s success in CVE-2016-9079 for ZEUS’s different randomization buffer sizes for memory allocations. The figure includes results *i)* for only the exploit’s first step before the `HandleContainerTimeChange` call and stack pivot launch (which is insufficient for the attack’s completion), and *ii)* considering all the aforementioned steps of the exploit that results very low success likelihood.

4.5 Complementing Isolated Heap

Microsoft developed isolated heap (IH) [101] in 2014 for Internet Explorer following emerging UAF attack. It stores each object in one of two isolated heaps depending on which function initiates the object. Most of the DOM objects are allocated in a different heap from strings and arrays that are commonly used in UAF exploits. IH’s objective

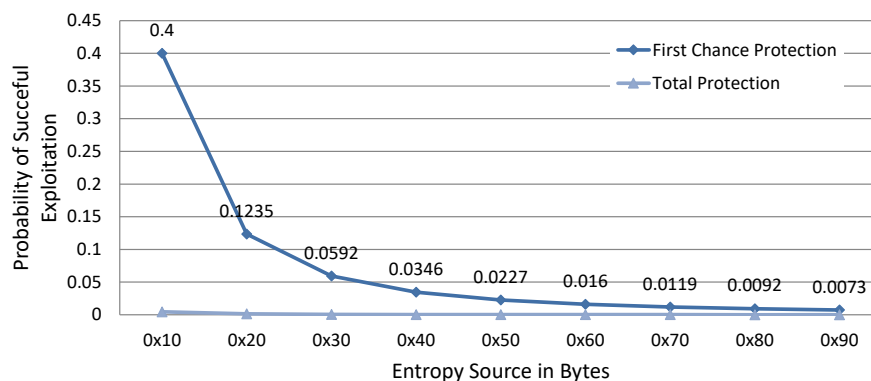


Figure 3.12: Exploit success rate for CVE-2016-9079.

Object MSHTML!CDOMTextNode

05ad6760	601b3e5c	MSHTML!CDOMTextNode::`vftable'
05ad6764	00000001	
05ad6768	00000001	
05ad676c	00000008	
05ad6770	00000000	
05ad6774	00000000	
05ad6778	08602420	
05ad677c	07215c90	
05ad6780	05ad4338	
05ad6784	05ad4338	
05ad6788	00000000	
05ad678c	071ade78	
05ad6790	40000000	
05ad6794	00000000	
05ad6798	1512471a	
05ad679c	0c009ca8	
...		

Figure 3.13: Internet Explorer CDOMTextNode object in memory. Object's offset 0x30 (boxed) is attacker-controllable memory address.

is to limit the options for the adversaries to reallocate a freed object. Later, Google and Adobe adopted very similar ideas for Chrome (PartitionAlloc [34]) and Flash [100].

IH leaves the heap layout deterministic (predictable). Additionally, its deployment requires detailed knowledge about the program source and manual effort to select a subset of data structures for isolated heap allocations. Recently IH was broken [2]. The exploit managed to replace a member field of pointer type in a recently freed CTableRow object with another CDOMTextNode object content (value: 0x40000000). Those two type of objects reside in the same heap by IH. The replaced and replacing values sit at the same offsets from the beginning of the corresponding objects. Due to the fine-grained determinism of IH, both objects were allocated at the beginning of the regions. This facilitated the successful exploit. The address 0x40000000 was a user model

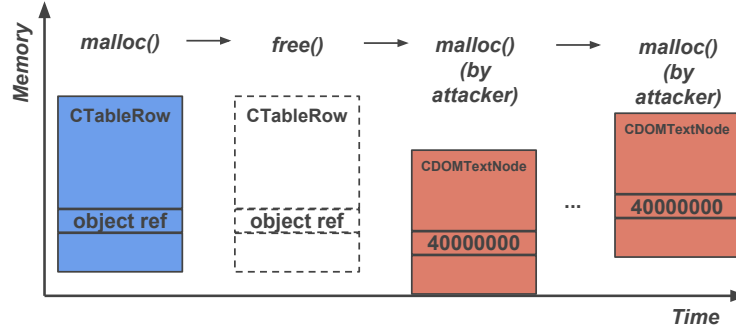


Figure 3.14: ZEUS prevents Microsoft IH UAF exploits.

attacker-controlled memory. It was already heap-sprayed using a maliciously invoked JavaScript function. Once IE browser dereferenced the original object’s member field to involve a member function, it launched the stack pivot as the starting point of the ROP attack.

As mentioned, an *exact* alignment between the original and the new object is required for the attack to succeed. Furthermore, heap-sprayed address 0×40000000 relies on deterministic memory layout. ZEUS’s protection targets both steps, and prevents controlled alignment and the deterministic heap layout. Figure 3.14 illustrates the defense.

5 Conclusion

We presented ZEUS, a defense mechanism against use-after-free (UAF) exploitations. ZEUS uses fine-grained non-determinism and implements a random heap allocator. It leaves the heap layout non-deterministic to the adversaries. Upon each allocation, the proposed defense introduces a random offset before each object on the heap. Hence, it prevents the attackers from implanting the target freed memory addresses precisely. According to our experiments, ZEUS was able to terminate the real past CVE exploits against large popular web browsers such as Firefox and Tor at their early stages within a negligible runtime performance overhead of 1.2% on average.

Chapter 4

Effective Mitigation for Kernel-Level Time-of-Check-to-Time-of-Use Vulnerabilities

1 Introduction

Time-of-check-to-time-of-use (TOCTOU), a.k.a race condition, or double fetch, is a long-standing issue in software security. As the name implied, it involves two references on the same variable or system state at different timing. The attacker usually passes the first security check with a benign value and then replace it with a malicious one before the second reference. This behavior could introduce faults into the system, therefore empower the attack ability to exploit the system.

This paper focuses on kernel-level TOCTOU vulnerability and mitigation. Modern operating systems such as Windows and Linux separate the kernel and user programs into two security domains. The kernel runs in the privileged domain, and the user programs run in the unprivileged domain. Due to the domain separation, the kernel serves the user programs on a service-client basis through system calls, and it unavoidably gets parameters from the userspace. Reading the same user-mode variable repeatedly by the kernel may lead to data inconsistency under a race condition between the kernel and user programs. In specific, the user program invokes a system call with the valid parameters and then replaces them with the malicious values after the security check passing. In this way, a user program can inject faults into the kernel, leading to the kernel malfunction or even running user-provided unsafe code. In particular, attackers could utilize such vulnerabilities to bypass sandbox protection of browsers or further get the administrator privilege after having a local user account.

By studying the real-world TOCTOU cases, we found that kernel-level TOCTOU vulnerabilities widely exist in the operating systems, especially Microsoft Windows.

Notably, a graphical subsystem kernel module tightly coupled with user-mode libraries freely access user data structures, among which double fetches on the same variable is not unusual. In essence, the kernel repeatedly reads the same user memory address within the same system call.

To find the bugs with the typical memory-access pattern, we develop TTFUZE, a fuzzing tool that leverages an Intel processor feature, Supervisor Mode Access Prevention (SMAP), and then combine it with a run-time hypervisor to monitor the Windows kernel efficiently. SMAP is introduced since the Intel Broadwell microarchitecture. It prevents the kernel from freely accessing userspace so that such access will raise an exception. It accurately serves the purpose of notifying us when the kernel access a user-mode address. For each system call, we pick out those user-mode addresses that the kernel reads twice, which are the candidates for further analysis. We contacted Microsoft for our findings.

Furthermore, we present SMAPRO, to the best of our knowledge, the first run-time protection for kernel-level TOCTOU vulnerabilities. It also leverages the same hardware feature SMAP and the hypervisor. Base on our observation of the vulnerabilities, to retain the virtual memory that the kernel fetched unwritable is the key to successful protection. Considering hardware capability and practical aspects, we propose the following idea. Whenever the kernel accesses a user-mode memory, SMAPRO protects the corresponding page by setting it as a kernel page until the current system call ends so that no other user threads can tamper with it. We also solve the practical issue that benign read and write to the data that resides on the protected page.

Due to the Windows system’s complex nature and the fact that it did not adopt SMAP, the amount of exceptions is enormous when enabling this feature, and the root causes are various. It is difficult to handle such a multifactorial situation, if not impossible. To first solve the core issue without interference from other factors, we use the hypervisor to confine SMAP within specific processes. The hypervisor takes action on the process context switch event and makes SMAP active only when the specific process is running on the processor. Later, we find that to prevent nested SMAP exceptions from forming a deadlock during page table walking, isolating SMAP is also crucial.

Additionally, it makes SMAPRO more configurable, avoids unnecessary processes, thus improves performance. Therefore, we consider the light-weight hypervisor framework as one of the contributions of this paper.

SMAPRO successfully mitigates real-world vulnerabilities such as CVE-2008-2252 and the family of CVE-2013-1254. It prevents the race condition from happening so that the system can operate normally during the attacks. We evaluate SMAPRO and the light-weight hypervisor with 18 benchmark programs and real-world applications. Our evaluation results show that SMAPRO imposes little extra overhead (less than 10% on average).

Contributions. To summarize, we make the following contributions in this paper:

- We identified kernel TOCTOU vulnerability using study cases and demonstrated with practical exploits against them.
- We present SMAPRO, a novel run-time mitigation framework leveraging hardware feature (SMAP).
- We propose a configurable light-weight hypervisor to isolate system-wide processor features.
- We develop a fuzzing tool TTFUZE for detecting kernel TOCTOU vulnerabilities.
- We have implemented SMAPRO and evaluated it with a number of benchmark programs with real-world vulnerabilities.

Roadmap The rest of this paper is organized as the following. In Section 2, we provide necessary background related to the mechanism behind kernel TOCTOU vulnerabilities and SMAP. Section 3 describes the objectives, threat model and scope, challenges and architecture of SMAPRO. Section 4 shows the vulnerability findings on Windows with our fuzzing tool, we present how we perform run-time mitigation with SMAP and a light-weight hypervisor (Section 5), with implementation details (Section 6), and evaluation (Section 7) respectively. In Section 8, we discuss an alternative to solve the writing conflicts and fuzzing methods, followed by related work in Section 9, we discuss related works. Finally Section 10 concludes the paper.

2 Background

2.1 Kernel-level TOCTOU Vulnerability

TOCTOU is a type of vulnerability that involves two references on the same variable or system state. The attacker usually passes the first security check with a benign value and then alters it to malicious before the second reference. It is a classic vulnerability, mostly existing among file system APIs. There are some previous research works have been working on addressing it [44] [23] [20] [19] [165].

Besides, TOCTOU also exists in the operating system kernel. Unlike the classic file system TOCTOU among APIs, kernel-level TOCTOU happens within individual system calls. When a user program invokes a system call, it usually needs to provide parameters. The kernel's responsibility is to verify the parameters' legitimacy and saves a kernel copy for subsequent use. However, the kernel may fail to accomplish that. The first reason is the developer's coding style or the unawareness of such vulnerability. The kernel routines could fetch the data from userspace instead of using the kernel copy. What is even worse is that for a historical reason, the kernel module may not fully decouple with the user-mode components; thus, it directly uses user-mode data.

The double or multiple fetches may lead to a severe issue, as we call it kernel-level TOCTOU vulnerability that attackers usually use to escalate privilege. The parameter sent into the kernel is benign initially to pass the security check; then, the attacker alters it to malicious to introduce an error such as a buffer overflow to the kernel. Although the time window between two kernel fetches may be as narrow as several instructions, it is feasible to create the race condition with careful craft, especially on a multi-processor system.

Due to mistakenly repeated operations on user data, kernel-level TOCTOU widely exists among operating systems [164] [169] [97], even in a system such as Linux that use particular gateway functions, `copy_to_user()` and `copy_from_user()`, to get user parameters [161]. Table 4.1 lists a portion of recent kernel-level TOCTOU vulnerabilities.

Table 4.1: Recent vulnerabilities categorized as race condition or time-of-check-to-time-of-use in the CVE database.

CVE-ID	Affected System	CVE-ID	Affected System
CVE-2008-2252	Windows	CVE-2016-5728	Linux
CVE-2013-1280	Windows	CVE-2016-6130	Linux
CVE-2018-7249	Windows	CVE-2020-9796	macOS
CVE-2020-9839	macOS	CVE-2020-9990	macOS
CVE-2016-10439	Android	CVE-2016-7624	macOS
CVE-2016-10383	Android	CVE-2017-7115	iOS
CVE-2020-5967	Nvidia	CVE-2020-8680	Intel

Figure 4.1 shows a Windows Win32k module’s kernel-level TOCTOU vulnerability [78] [144], which has been identified and patched in Microsoft security bulletins ms08-061. The pseudo-code reassembles to a Win32k system call, and the red part shows the vulnerable data’s trace. The user program passes `lParam` to `win32k_function()` through upper layer APIs. Adding `my_struct->cbData` to `cbCapture` is where the kernel first gets this user-mode variable and allocates a buffer based on its value. Notice, although there is a local variable called *capture*, the developer forgot to use it subsequently. Several instructions after, the kernel reread this user-mode variable when copy data into the new buffer. An attacker can alter the user-mode variable `my_struct->cbData` between the two reads, and especially by enlarging it, a kernel buffer overflow is created.

Figure 4.2 gives the assembly-code-level exploit simulation. As a local privilege escalation vulnerability, the attacker can invoke the vulnerable system call as many times as needed to succeed. Meantime, thread one is spawned to race with the kernel, aiming to enlarge the user variable. To generate the buffer overflow, the attacker needs to flip the high bit an odd number of times during the time window.

2.2 Supervisor Mode Access Prevention (SMAP)

Monitoring the kernel’s userspace behavior is essential to SMAPRO. Due to x86 protected mode characteristics, there is no mechanism available for a broad range of monitoring memory modifications. Techniques such as leveraging hardware watchpoints or transactional memory are fittable for fuzzing such vulnerabilities but not for run-time

```

// lParam points to data located in user space
void win32k_function(... LPARAM lParam, ...)
{
    DWORD cbCapture;
    ...
    my_struct = (PMY_STRUCT)lParam;

    // first fetch
    cbCapture = sizeof(MY_STRUCT) + my_struct->cbData;
    ...
    pNew = UserAllocPoolWithQuota(cbCapture, AG_SMS_CAPTURE)
    if (pNew != NULL)
    {
        RtlCopyMemory(pNew, my_struct, sizeof(MY_STRUCT));
        // second fetch
        RtlCopyMemory(pNew, my_struct->lpData, my_struct->cbData);
    }
    ...
}

```

Figure 4.1: Pseudocode of the vulnerability fixed in ms08-061. The vulnerable variable is in red. The kernel reads it twice, and it may get a different value for the buffer allocation and the subsequent buffer copying. It is common to see such a coding style. However, it is vulnerable because the two reads cross the privilege boundary.

protection. We discuss these two techniques in more details in Section 4 and Section 9.

Fortunately, we notice an Intel processor feature so-called Supervisor Mode Access Prevention (SMAP) [37] that accurately serves our kernel monitoring requirement. SMAP is a feature that prevents the kernel from freely accessing userspace so that such access will raise an exception. It complements Supervisor Mode Execution Prevention (SMEP) [52] that introduced earlier. SMEP can be used to prevent the kernel from unintentionally executing user-mode code. SMAP extends this protection to reads and writes. It makes it harder for a malicious program to deceive the kernel into using code or data from the userspace.

Setting SMAP (20) bit in CR4 enables it, and it can also be temporarily disabled by setting the AC flag in EFLAGS or through using `stac` and `clac` instructions. Temporarily disabling SMAP usually indicates that the kernel is fully aware of its userspace-access behavior. For example, The Linux kernel supports SMAP since version 3.7. The kernel-to-userspace accesses must go through two gateway functions `copy_to_user()` and `copy_from_user()`, in which SMAP is temporarily disabled.


	(Thread 0)		(Thread 1)
	...		mov eax, pcbData
	mov [eax+var_48], eax		...
	<u>mov ebx, [eax+41]</u>		...
	add ebx, 0Ch		...
	mov ebx, [ebp+var_24], ebx		xor [eax], 0x80000
	test ebx, ebx		xor [eax], 0x80000
	jnz bailout		xor [eax], 0x80000
	push 1		xor [eax], 0x80000
Attack	push 63737355h		xor [eax], 0x80000
Time	push ebx		xor [eax], 0x80000
Window	call		xor [eax], 0x80000
	UserAllocPoolWithQuota		xor [eax], 0x80000
	...		xor [eax], 0x80000
	mov [eax+8], edi		xor [eax], 0x80000
	mov eax, ecx		...
	<u>mov ecx, [eax+41]</u>		...
	mov esi, [eax+8]		...
	mov eax, ecx		
	shr ecx, 2		
	rep movsd		

Figure 4.2: Thread zero invokes the vulnerable system call. The attack windows lie in between the two kernel reads, namely, the two instructions that underscored. The attacker can repeatedly open the attack time window by calling the system call. Simultaneously, thread one flips the high bit of the user-mode variable in a loop. The two threads compete, hoping an odd number of flips occur during the time window. It enlarges the variable; otherwise, the variable remains the same.

However, Windows does not support SMAP still. The kernel takes a different approach other than gateway functions; it uses *probe* and *capture* in each system call. `ProbeForWrite()` and `ProbeForRead()` [109] validates user-mode variables and buffers and this method is effective if done correctly and thoroughly. However, kernel components such as Win32k failed to follow the coding rule. Some of its code is still coupled with user-mode components. It will cost a huge engineering effort to change the coding style for the enormous codebase.

2.3 Intel Virtualization Technology

Intel Virtual-Machine Extensions (VMX) provides hardware-assistant virtualization that adds 13 new instructions: `VMPTRLD`, `VMPTRST`, `VMCLEAR`, `VMREAD`, `VMWRITE`, `VMCALL`, `VMLAUNCH`, `VMRESUME`, `VMXOFF`, `VMXON`, `INVEPT`, `INVVPID`, and `VMFUNC`. VMX has root and non-root mode, where root mode runs the hypervisor, and the non-root mode runs virtual machines or called guests. On x86 architecture, the processor has four privilege rings. The kernel runs at ring zero, the

highest priority ring; the user programs run at ring three; ring one and ring two are unused. With VMX, the root model is commonly viewed as the ring minus one, which is more privileged than ring zero.

VMXON/VMXOFF enters/exits VMX mode. The Virtual Machine Control Structure (VMCS) is the most important data structure, storing the data and states of one virtual processor of one virtual machine. Each core in a physical processor has a VMCS pointer. It points to the physical address of the VMCS. VMPTRLD loads the VMCS pointer from physical memory and set it active and current. VMCLEAR stores VMCS active states back to memory and set it inactive. Although the hypervisor is fully aware of the physical address of each VMCS, it can not modify them directly. All the modifications on VMCS should use the instruction VMREAD and VMWRITE instead.

The VMCS manages aspects of a virtual machine using many data-fields organized into six logical groups: Guest-state area, Host-state area, VM-execution control fields, VM-exit control fields, VM-entry control fields, and VM-exit information fields. The last four groups compose VMX controls that conduct the virtual machine's behavior. In VMX's term, VM entry is the transition from VMX root mode to the non-root mode; VM exit is the opposite. During a VM exit, the processor stores its running state into the Guest-state area in memory and loads the Host-state area into the hardware. The processor does the opposite when entering a virtual machine, the so-called VM entry.

2.4 x86 Architecture

In this part, we briefly describe concepts and mechanisms regarding x86 architecture that are used in SMAPRO. Some techniques are rarely mentioned in researches but vital to SMAPRO. Although IPI and APIC may not technically belong to x86 architecture concepts, we put them here because they are related to the underlying hardware mechanisms.

Paging and Virtual Memory. On x86 architecture, with the flat or the segmented memory model, linear address space is mapped into the processors' physical memory space either directly or through paging. Direct mapping is a one-to-one mapping between the linear address and physical address, also known as the real-mode.

When using the paging mode, the linear address space (often referred to as virtual memory) is composed of pages. For simplicity, we only consider 4KB pages in this paper. The pages are backed with physical pages through the Memory Management Unit (MMU). This hardware component automatically translates virtual addresses to physical addresses using a data structure called the page table. The translation creates the illusion for each process that it has its own large flat virtual memory space (4GB on a 32-bit system).

Page Table. The MMU uses page tables to map physical pages to virtual pages [70] so that each process in the system can have a flat virtual memory space. Because each process has a page table, the memory used to store those page tables is unignorable. Therefore the system designs a hierarchical data structure to save physical memory. As shown in Figure 4.3, a virtual address splits into three parts. The last 12-bit is the byte offset of the page; the first two 10-bit are the index to the page directory and page table. The page tables are also composed of 4KB pages where the system swaps out the long-time unvisited pages to save physical memory. When walking through the page table, we often encounter a swap-out page, which is not an issue because it will be automatically brought back by a page fault regarding page absence. However, this becomes a problem since we are already in the context of a SMAP page fault. The page swap-in process needs the system to read the hard disk, thus more system calls, which inevitably trigger more SMAP exceptions. Eventually, they may cause a dead loop. Therefore, our hypervisor-based solution that confines SMAP at the process level is essential to solving this issue.

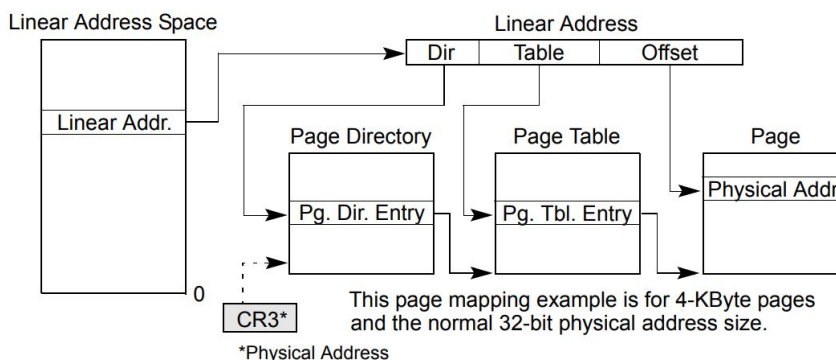


Figure 4.3: Linear-Address Translation to a 4-KBbyte Page using 32-Bit Paging [70]

Translation Lookaside Buffer. As mentioned above, page table walking is a lengthy process. A full walk needs to access two pages (PD page and PT page). If any of the two pages are not present in the memory, it further triggers a page fault to bring the absent page back.

A Translation Lookaside Buffer (TLB) is part of MMU, and it is a memory cache used to reduce the time to access virtual memory. The TLB has a fixed number of slots that stores the recent translations of virtual memory to physical memory and permission bits. If a valid TLB entry exists, the processor will not walk the page table trying to find the Page Table Entry (PTE). The system kernel is responsible for the consistency between TLB and the page tables.

Interrupt Descriptor Table. The Interrupt Descriptor Table (IDT) is a data structure used on the x86 architecture to implement an interrupt vector table. The processor uses the IDT to determine the correct response to interrupts and exceptions.

Interrupt and Exception. The fundamental difference in microarchitectural between interrupt and exception is as follows. An interrupt is an asynchronous event that is typically triggered by an I/O device. An exception is a synchronous event generated when the processor detects one or more predefined conditions while executing an instruction. However, one thing in common is that their handlers are all in the IDT, making them easily confused.

Furthermore, exceptions are categorized as **faults**, **traps**, and **aborts** depending on the way they are raised and whether the instruction that caused the exception can be restarted without loss of program or task continuity. Aborts are not recoverable. They are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables. Faults and traps are recoverable. The main difference between them is that when recovers from faults, the return address is the faulting instruction; the trap's return address points to the instruction after the trapping instruction. In our case, the SMAP exception is a fault and handled through the page fault handler.

The IDT has 256 entries. Entry 0-31 are for exceptions, except entry two is for Non-Maskable external Interrupt (NMI). The rest is for external interrupts from INTR pin or INT n instruction. However, INT n also known as a software interrupt. It is

essentially an exception with *interrupt* in its name, and its vector located with other external hardware interrupt vectors. Fun.

Trap Frame. The trap frame is a data structure that the processor pushed into the kernel stack when an interrupt or exception occurs. It contains part of the faulting thread's registers. In the context of a page fault, the trap frame contains ErrorCode, CS: EIP, EFLAGS, and SS: ESP. The SS: ESP is included based on whether the CS register's current privacy level (CPL) changes.

Gate. Code modules in lower privilege can only access higher privilege modules through a tightly controlled and protected interface called a **gate**. There are four types of gate, namely, **task gate**, **trap gate**, **interrupt gate**, and **call gate**. Fully describe and distinguish them is out of the paper's scope. The project only involves the interrupt gate.

The vectors in the IDT go through either the interrupt gate or trap gate. The difference is subtle. Suppose the interrupt or exception handler is called through an interrupt gate. In that case, the processor clears the interrupt flag (IF) in the EFLAGS register to prevent subsequent interrupts from interfering with the handler's execution. On the other side, when through a trap gate, the processor does not change the IF flag. Setting the interrupt flag is a requirement to control stack depth, which also involves the type of interrupt controller installed in the system. We observed that the processor invokes the page fault handler through an interrupt gate, and we think the reason for that is as follows. The processor loads the CR2 register with the faulting virtual address that caused the exception. If the interruption is allowed in the processor, a thread context switch may swap out the page fault handler, and another page fault occurs subsequently and overwrites the CR2.

CS Segment and Current Privilege Level. Unlike other segment registers, the CS segment register cannot be set directly using instruction such as MOV, but only through a trap or an interrupt gate. The processor maintains the Current Privilege Level (CPL) field in the CS segment. Thus it is always equal to the processor's current privilege level. The CS register of the trap frame provides us the privilege of the code when the SMAP exception happens. Therefore, we do not make decisions based on

EIP’s apparent value, making the mitigation more reliable.

Inter-processor Interrupt. It is an interrupt controller mechanism to interrupt another processor or group of processors on the system bus. They are used for software self-interrupts, interrupt forwarding, or preemptive scheduling. In SMAPRO, we use IPI to flush the TLB cache on all the processors.

Inter-processor Interrupt. It is an interrupt controller mechanism to interrupt another processor or group of processors on the system bus. They are used for software self-interrupts, interrupt forwarding, or preemptive scheduling. In SMAPRO, we use IPI to flush the TLB cache on all the processors.

Local APIC. Intel’s Advanced Programmable Interrupt Controller (APIC) is a family of interrupt controllers. Nowadays, multi-processor systems utilize APIC instead of the obsoleted Intel 8259 Programmable Interrupt Controller (PIC). The APIC is a split architecture design, with a local APIC integrated into the processor and the I/O APIC on the system chipset. The local APIC receives interrupts from various sources such as I/O APIC then sends them to the processor, and it also receives/sends IPI messages from/to other processors on the system bus. On the other side, the I/O APIC is responsible for receiving interrupts generated by I/O devices and forwarding them to the local APIC.

A processor can program the local APIC’s interrupt command register (ICR) to send out IPIs. The target local APIC receives the IPI message and calls the processor’s IDT vector according to the information that comes with it, such as the vector number. Local APIC registers are memory-mapped to a 4-KByte region of the processor’s physical address space with an initial starting address at 0xEFF00000. The system software interacts with it through memory operations.

3 Overview

3.1 Threat Model

Kernel TOCTOU is a local privilege escalation vulnerability. The vulnerability could allow local users or malicious software to gain full root privileges. We assume that

an attacker has a user account that can upload and run arbitrary programs with user privilege or access such a program. The attacker has arbitrary memory reads and writes primitives. He is also able to call any system service or load any library. The DEP policy (**W**rite \oplus **eX**ecute) and ASLR is not necessary. We assume the attacker has full knowledge about the system kernel, including the memory layout. However, he can not read or write any kernel memory because a classical operating system would not allow it. The attacker aims at running arbitrary code in the kernel, hence obtains the highest privilege.

We endeavor at the Windows OS, a complicated operating system. Linux kernel is out of scope. Considering we leverage a hardware feature from the Intel processor, The host system needs to have an Intel processor with SMAP capability.

3.2 High-Level Design and Challenges

The kernel double fetching a user address may cause a TOCTOU vulnerability. There are two ways to prevent exploitation; one is to drop the kernel's double-fetch action; the other is to prevent data mismatch between fetches. From a practical point of view, we can not decline the kernel's double-fetch behavior. Therefore, this paper proposes a framework to prevent data mismatch between fetches. The high-level idea is that when the kernel accesses a user address, we freeze the containing page so that no other user thread can overwrite it.

However, there have been many challenges here:

- How can we know when the kernel accesses a user address?
- It is overkill to freeze an entire page just for one variable and reads should allow.
- The two fetches should happen within a short time, more specifically, within the same system call.
- Windows is a complex operating system. How can we practically enable a system-wide hardware feature such as SMAP without crashing the system?

3.3 Approach Overview

How to get notified when the kernel accesses a user address is the biggest challenge. Since the processor reading memory is such an ordinary operation, no official hardware feature is available for monitoring it in a broad range of memory. To solve this challenge, we abuse a hardware feature SMAP. Originally, SWAP is designed to prevent the attacker from tricking the kernel into getting shellcode or malicious data from userspace. However, one unique characteristic of this feature is that when the kernel accesses a user address, the processor raises a page fault exception. This feature accurately serves our purpose so that we will leverage it in a novel way to solve the challenge.

Subject to the x86 architecture, the protection has to base on the page granularity. To protect even one byte, we have to protect an entire page. First, we separate the subsequent user reads from writes because the reads are harmless. When we temporarily release a page, we set this page as read-only to ensure no writing. When a user writes on the protected page, we make the thread suspend on the writing instruction until the current system call ends.

As previously mentioned in Section 2, the kernel-level TOCTOU vulnerabilities happen inside individual system calls. We hook Windows internal functions to know when a system call ends and releases the pages it accessed. Also, we monitor the creation and termination for both processes and threads and use Windows internal data structures to distinguish each thread.

We develop a light-weight hypervisor to confine the SMAP feature into specific processes. It makes debugging less painful to us, and it is also necessary to prevent nested SMAP exceptions that cause a dead loop.

Figure 4.4 shows the high-level overview. The hypervisor enables SMAP only in one process, which is the one currently running on the processor. The kernel has accessed three user pages, as marked in the user process memory space, and one user thread in the same process tries to access those pages. Notice the read is allowed. The protected page is temporarily released and set read-only. However, the write is not permitted for the moment; the page fault handler suspends the thread to avoid causing any data

inconsistency in the kernel.

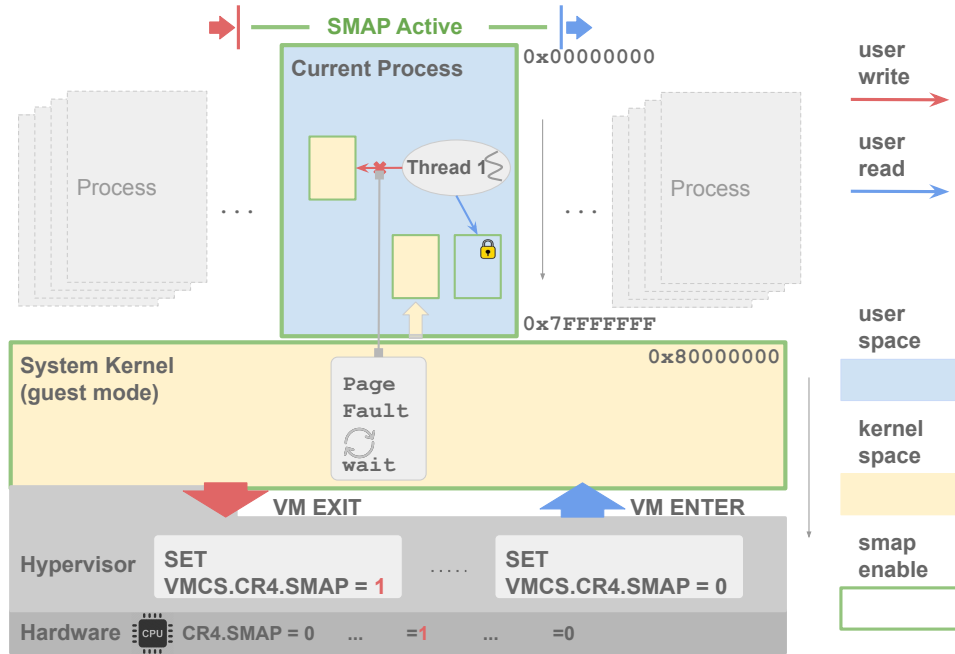


Figure 4.4: The hypervisor is capable of confining the system-wide feature SMAP into one process. When the hypervisor catches the process context switch events, it changes the SMAP enable bit in the CR4 register to only set during the target process. The processor raises page fault exception when the kernel accesses userspace so that we can protect those pages. Thread 1 can read a protected page but can not write. The read is allowed by automatically setting the protected page back to userspace with a read-only permit. However, when the write instruction raises an exception, the page fault handler suspends the thread until the system call ends.

4 Finding Kernel TOCTOU Bugs

This section presents our fuzzing tool TTFUZE for kernel-level TOCTOU vulnerability and the results we obtained on Windows, especially from the Win32k graphical subsystem.

Kernel-level TOCTOU vulnerabilities are subtle. It is not intuitive for the developers to be aware of those double fetches. They are not apparent errors such as buffer overflow that the compiler-based method [118] [175] can detect. It is even harder to spot than the use-after-free vulnerability that code emulation or run-time checks can expose [51] [30] [92] [143]. As described in Section 2, the exploit needs to win the race with the kernel in high speed, and only by chance that it can trigger the vulnerability.

From previous research works and vulnerabilities' publicly available information, the cause of kernel-level TOCTOU vulnerability is either due to historical reasons that a kernel module is highly coupled with user libraries or because of sloppy coding style.

However, this vulnerability has a solid memory access pattern, namely, two reads from kernel to userspace. It is not convenient to observe such a pattern either from user programs or the kernel. The most intuitive way is to use a virtual machine. Previous work such as [78] [77] utilize a full software emulated virtual machine Bochs [90], which parses and executes each instruction without hardware virtualization assistant or dynamic binary translation. It is straight forward to instrument the operating system kernel, and it discovers many double-fetch vulnerabilities [78] [77]. However, due to the nature of full software emulation, the virtual machine runs extremely slow. So it is not easy to make a comprehensive test, especially with GUI programs, because GUI needs a prompt response.

As previously described in Section 2, the SMAP feature is an ideal way of monitoring kernel-to-user-memory behavior. The goal is to check if the same address is being read twice by one system call.

However, the difficulty lies in recovering the system from a fatal SMAP exception. Because of its original intention, the operating system should crash when it receives such an exception. We tried different methods to cancel a SMAP exception, such as disabling it at CR4.SMAP or temporarily disable it through setting EFLAGS.AC. None of them works. Fortunately, setting the faulting page to kernel-mode is one way to satisfy the processor so far. Subsequently, we want to release the page back to user-mode as soon as possible not to lose track of the kernel.

Single Step Trap. It is the soonest way to get back control. Through setting the trap flag (TF) in the EFLAGS register, the processor stops at every instruction. After setting the faulting page to kernel-mode to recover from the SMAP exception, we want the processor to stop at the next instruction. So we can release the protected page and continue the kernel.

For the single-step trap, the processor automatically sets the resume flag (RF) in the EFLAGS register so that the handler itself will not be interrupted on every

instruction. Because we use a hypervisor, the single-step trap triggers a VM exit. In the event handler, we try to clear the TF or set the RF in the guest EFLAGS and then resume the virtual machine. However, the RF flag does not work properly under such circumstances; the single-step still comes in.

Breakpoint We decide to take an alternative approach that writes a software breakpoint directly on the next instruction. We first parse the current instruction’s length and then write a byte 0xcc to it. Next, we record the information of this access and let the kernel continue. When the hypervisor gets the debug trap, we fix it with the original byte, release the protected page, and then resume the virtual machine to re-execute the faulting instruction.

Results. We found a few double fetch candidates; Table 4.3 in Section 7 gives a glimpse of the problem. This method has advantages over the binary static analysis. In the x86 instruction set, there are a few addressing modes of accessing memory, such as *absolute*, *register*, *register + offset*, which makes it difficult to identify the same address from the syntax perspective. For example, in Figure 4.5, it needs to trace both ECX and EDX registers and, in some cases, need to get user input to calculate the final address eventually. In TTFUZE, each user address visited is reported through the hardware mechanism SMAP. Therefore, it can accurately locate every double fetch within a system call.

bf812ddf 8b03	mov	eax,dword ptr [ebx]	
bf812de1 8b502c	mov	edx,dword ptr [eax+2Ch]	
bf812de4 8bb284010000	mov	esi,dword ptr [edx+184h]	
bf812dea 8b9288010000	mov	edx,dword ptr [edx+188h]	(0x4808c4)
bf812df0 899518ffffff	mov	dword ptr [ebp-0E8h],edx	
bf812df6 8b511c	mov	edx,dword ptr [ecx+1Ch]	
bf812df9 3bf2	cmp	esi,edx	
...			
bf812e42 8b482c	mov	ecx,dword ptr [eax+2Ch]	
bf812e45 8d8184010000	lea	eax,[ecx+184h]	
bf812ebe 6683790203	cmp	word ptr [ecx+2],3	
bf812ec3 0f84e1ffffff	je	win32k!GreBatchTextOut+0x3d (bf812daa)	
bf812ec9 898500ffffff	mov	dword ptr [ebp-100h],eax	
bf812ecf 8b5120	mov	edx,dword ptr [ecx+20h]	

Figure 4.5: CR3 0x6d40320; TEB 0x7ffdd000; EIP 0xbf812de4 and 0xbf812e4b read the same user-mode address 0x4808c4 within one system call.

Whether those bugs can further become vulnerabilities is need to analyze case by case. However, a significant potential of those accesses is not within a try-catch block. It may lead to a local denial-of-service (DOS) attack if the attacker frees the user page right ahead of the access.

5 SMAPro Design

This section presents the design of SMAPRO. The core of SMAPRO includes two key components, the system module, and the hypervisor. The system module intercepts the system’s page fault handler to process SMAP and relevant exceptions, which is the protection’s main logic. The lightweight hypervisor puts the system into the virtual machine. Its primary use is to confine the system-wide feature SMAP into specific processes.

5.1 System Module

The system module’s core functions include enabling SMAP, recovering from SMAP exceptions, protecting pages, and solving read/writes conflicts. We describe each technique as follows.

Monitoring Kernel to Userspace Access. The biggest challenge we confront is how to monitor the kernel-to-userspace behavior efficiently. Accessing memory is such an ordinary operation so that no purposeful hardware feature is available for monitoring that. We notice a rarely mentioned hardware feature SMAP. Its initial design prevents the attacker from tricking the kernel into getting shellcode or malicious data from userspace. When the kernel accesses a user address, the processor raises a page fault exception. This part accurately serves our purpose, so we want to leverage it in a novel way.

However, not every aspect of this feature perfectly fits our exceptions. If in an ideal situation, the hardware should report on each kernel-to-userspace access and freeze the user memory at machine word granularity. In reality, SMAP only works on the page level, and the exception that it raised is fatal to the system, meaning the operating

system should crash when it receives such exceptions. We eventually find a way to recover it. We intercept the system's page fault handler to handle such exceptions. Since Windows does not support SMAP, we do not pass those exceptions to the kernel. Considering the violation of raising a SMAP exception is that the kernel accesses userspace, so puts the corresponding page into kernel space does the opposite, thus solve the violation. Otherwise, it is then too late to disable SMAP through CR4 or EFLAGS.AC.

Putting a page into the kernel not only solves the exception but also protects the page. The user threads no longer can access it, which prevents the race condition between the kernel and user threads. However, it is overkill to protect the entire page and block benign reads and writes on the rest of the data. We will elaborate on the read/writes conflicts in the following sections.

Solving Read Conflicts. For practical purposes, solving read conflicts is essential. It is common to have multiple global variables or multiple heap buffers share the same page. Therefore, when we protect an entire page, we block benign access to the rest of the data. It is especially unnecessary because reads do not harm security.

We solve the read conflicts by setting the protected page back to userspace, allowing user threads to read. When user threads read a kernel-mode protected page, the processor raises an exception due to the privilege violation. Therefore our page fault handler gets the notification and sets the page back to userspace. Additionally, the page is also set with a read-only permit to ensure no write. Figure 4.6 shows the transitions between kernel-mode and user-mode. We record the original page information to handle various situations and correctly restore it at the end of the system call.

Page Attribute Transition. The modifications on the page attributes are essential to this mitigation. Because changing a user page to kernel-mode is the primary method of protection. Moreover, to solve reads conflicts, we need to change the page back and forth between kernel-mode and user-mode.

First, we need to locate the page's Page Table Entry (PTE). As mentioned in Section 2, with paging, every page in the virtual memory has an entry in the page table. As shown in Figure 4.7, the User/Supervisor decides whether this is a kernel page or a

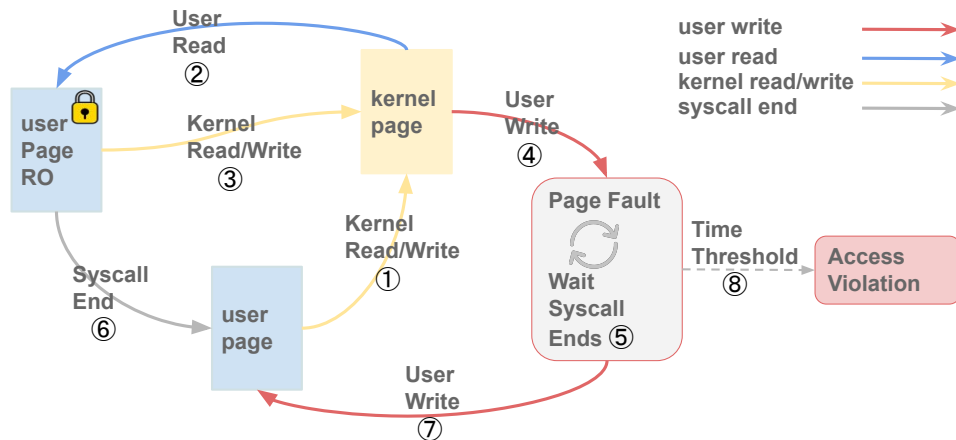


Figure 4.6: Page attributes transits in different states. A user page becomes a kernel page when the kernel read/write it ①. Afterward, if user threads read this page, our page fault handler changes it back to user-mode with a read-only permit ②. This page is again capable of triggering a SMAP exception if the kernel reaccesses it ③. Our page handler suspends any user thread that tries to write a protected page ④. It then calls a sleep function, letting the operating system, and wakes up periodically to check the page's status ⑤. When the current system call ends, this page is restored to user-mode with its original permits ⑥. The write thread is also released and re-execute the faulting instruction to write the page ⑦. However, if the thread waits too long, it will be terminated to avoid a deadlock ⑧.

user page where set if a user page, otherwise a kernel page.

Address of 4KB page frame	Ignored	G	P A T	D	A	P C D	PW T	U / S	R / W	1	PTE: 4KB page
Ignored										0	PTE: not present

Figure 4.7: **Bit 0 (Present): 0** indicates an invalid page. **U/S (User/Supervisor): 0** user-mode accesses are not allowed to the page referenced by this entry. **R/W: 0** writes are not allowed to the page.

We obtain various information in the context of the page fault handler to find the corresponding PTE. The CR2 register stores the faulting virtual address. Regarding SMAP, it is the user address that the kernel accessed. The CR3 register stores the physical address of the current page table base. The trap frame in the kernel stack contains the error code, CS: EIP, SS: ESP, and EFLAGS, which describes the processor context when the exception happens.

Changing the U/S bit in the PTE makes the user page becomes a kernel page. It is counterintuitive because we usually have the impression that the virtual address

between 0x80000000 to 0xFFFFFFFF is the kernel space on Windows 32-bit system. However, the processor mechanism defines the kernel space as follows. There is the Current Privilege Level (CPL) field in the CS segment. The processor maintains this 2-bit field to equal the processor's current privilege level. Meantime, the U/S bit in the PTE decides whether the unprivileged code can access it. Traditionally, we consider the memory space that only the most privileged code (CPL:00) can run as the kernel space. Indeed, it is a considerably complicated mechanism involving more data structure in the processor's microarchitecture, out of this paper's scope. In essence, the U/S bit decides whether the page is a kernel page or a user page, even the virtual address is below 0x80000000.

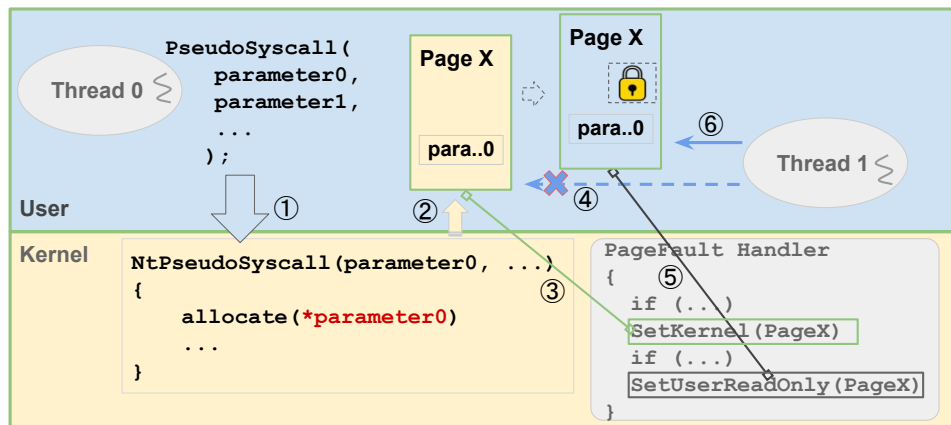


Figure 4.8: ① User thread 0 invokes a system call with parameters. ② The kernel fetches one of the parameters from userspace hence triggers a SMAP exception. ③ Our page fault handler converts the page into kernel-mode to protect it. ④ User thread 1 tries to read the protected page and triggers an exception due to privilege violation. ⑤ Again, our page fault handler processes the exception and converts the protected page back to user-mode with a read-only permit. ⑥ The faulting instruction re-execute, and the user thread successfully read the data without knowing the previous exception.

Figure 4.8 shows the mitigation convert a low address user page into a kernel page. It is a common situation where user programs invoke a system call and provide parameters. After that, if a user thread accesses a protected page, the processor raises an exception due to the privilege violation. The page fault handler converts this page to user-mode and read-only.

Solving Write Conflicts. We can not allow any user code to write a protected page during its protection, because unlike reads, the write operation is a security threat.

Subject to the x86 architecture, the protection has to base on the page granularity, so not all the writes on this page will cause a TOCTOU problem. Therefore, we can not directly terminate every user thread that accesses a protected page from a compatibility perspective. To solve this, we choose to delay the write operation. After the writing instruction caused an exception, our page fault handler suspends the user thread until the end of the system call. Therefore, the protected page remains the same for the kernel, and user threads can also write it after the system call. We explain the practicality of suspending a thread in the context of page fault in the following section.

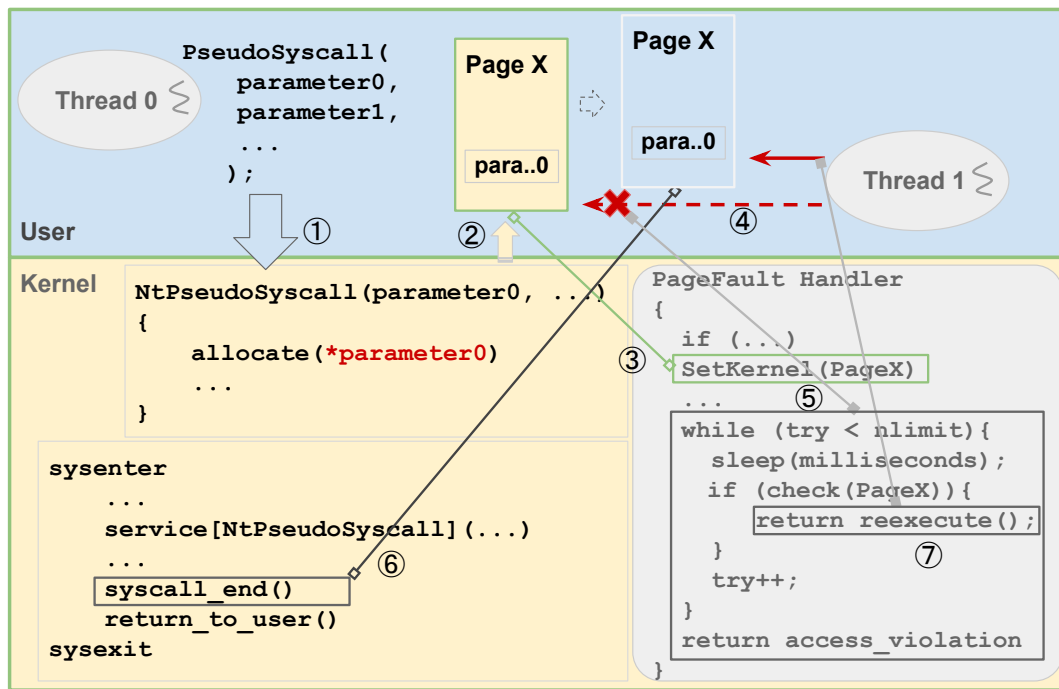


Figure 4.9: To solve this, we choose to delay the write operation. ① User thread 0 invokes a system call with parameters. ② The kernel fetches one of the parameters from userspace hence triggers a SMAP exception. ③ Our page fault handler converts the page into kernel-mode to protect it. ④ User thread 1 tries to **write** the protected page and triggers an exception due to privilege violation. ⑤ This time, our page fault handler suspends the thread by calling a sleep function, letting the operating system schedule. It rechecks the page's status periodically when it wakes up. ⑥ When the current system call ends, the mitigation releases all the protected pages related to this thread. ⑦ Meantime, the page fault handler wakes up and realize that the page is released. Therefore it finishes the exception by executing the faulting instruction again.

Figure 4.9 shows that thread zero invokes a system call with parameters. The kernel gets the user parameter so that the corresponded page is protected. Afterward,

user thread one tries to write the page, which raises a page fault exception due to privilege violation. The page fault handler suspends the current thread by calling a sleep function, namely, `KeDelayExecutionThread()`. The thread wakes up periodically to check whether the page is released. If so, the exception is finished and re-executes the faulting instruction. Otherwise, the page fault handler may terminate the thread to prevent a deadlock if it takes too long.

Interrupt and Exception. Suspending a thread inside the page fault handler seems unusual because the page fault exception resides in the Interrupt Descriptor Table (IDT) with interrupts, and they seem to be time-critical routines. However, there is an essential difference between an exception and an interrupt. An interrupt is an asynchronous event that is typically triggered by an I/O device. An exception is a synchronous event generated when the processor detects one or more predefined conditions while executing an instruction. Interrupts have a higher priority than the operating system's scheduler and most of the kernel components. Any job that takes too much time should not be processed in an interrupt handler [108]. On the contrary, exceptions have the lowest priority in the kernel. In Windows' term, the Interrupt Request Level (IRQL) that the exception handler executes at is `PASSIVE_LEVEL`, meaning call for thread scheduling is plausible.

Releasing Protected Pages. To be aware of when a system call ends, we choose to intercept Windows internal functions. We keep tracking the page table base and Thread Environment Block (TEB) to distinguish each thread, thus release the protected pages on a thread basis.

Flushing TLB. We need to flush Translation Lookaside Buffer (TLB) to ensure that the page attributes modification is effective on all processors of the system. TLB is a memory cache that stores the recent translation of virtual memory to physical memory. It accelerates the process of accessing virtual memory. Different than data cache, TLB is not entirely transparent to the operating system. When the operating system updates a page table, the corresponding TLB entries need to be invalidated to get new ones. As mentioned above, we leverage the page attribute transition as the main page protection. It is critical to ensure the PTE modification takes effect

instantly, especially on a multi-processor system. We examine the method to flush the TLB through local APIC as described in Section 2. Eventually, we find Windows internal functions that flush TLB entries on all the processors. Therefore we use them and do not have to consider the underlying hardware differences.

5.2 Hypervisor

The hypervisor plays an essential role in developing and debugging for SMAPRO. When we first enabled SMAP in Windows, instantly, an enormous amount of exceptions flooded the system. The debugger was frozen. It is not surprising because we know that SMP is a system-wide feature, and Windows does not support it. A significant portion of the system calls fetches user-provided parameters and system data such as Process Environment Block (PEB), `USER_SHARED_DATA` mapped in userspace, which all trigger SMAP exceptions.

Due to Intel VT virtualization technology’s design [120], it is possible to load a light-weight hypervisor as a kernel module during run time. Unlike other commercial hypervisors such as Xen, Hyper-V, and VMWare, it does not emulate hardware devices. It merely put the operating system into VM guest mode, and itself becomes the hypervisor, thus monitors system events [157].

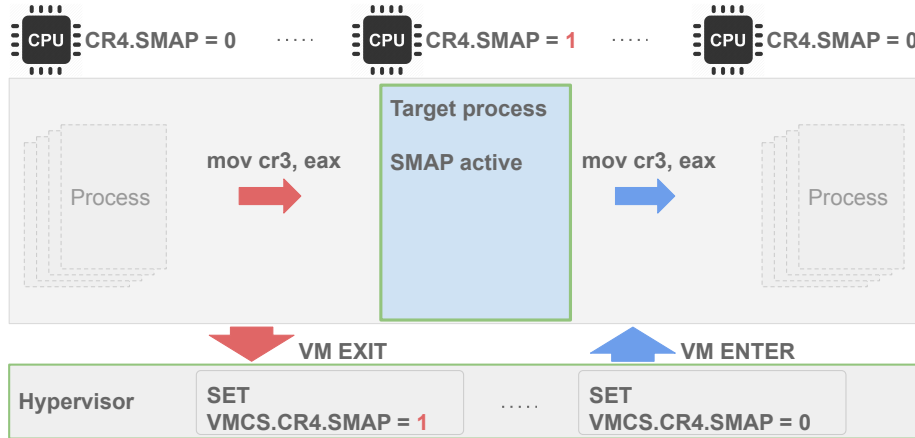


Figure 4.10: Operations on the CR3 register are the decisive characteristic of process context switching, which triggers VM exit by default. By setting the SMAP bit in the CR4 image of the VMCS.GuestArea, it updates the CPU CR4 when the hypervisor enters the virtual machine again. Therefore, it is possible to use the hypervisor to enable the SMAP feature when a specific process is running on the CPU.

By monitoring the process context switch event, namely, operations on the CR3 register, the hypervisor can temporarily enable/disable SMAP to make it only effective in the context of specific processes. Figure 4.10 shows that `mov cr3, eax` triggers a VM exit event, and the hypervisor receives it. If the new CR3 belongs to one of the target processes, the hypervisor sets the CR4.SMAP in the Virtual Machine Control Structure (VMCS), which is the data structure that updates the real CPU registers when entering the virtual machine. After returning to the guest virtual machine, the SMAP is active. When this process switches out, the hypervisor again receives the event and unset CR4.SMAP. Therefore, this particular process has the illusion that SMAP is active in the system while other processes feel the opposite.

The hypervisor inevitably brings performance overhead. However, it makes the mitigation more configurable. Due to the nature of local privilege escalation attacks, system processes that already have high privilege are not threats. Therefore it is not very meaningful to protect them, which reduces the overall performance overhead. Additionally, as previously mentioned, SMAP confinement is also necessary to prevent deadlock caused by nested SMAP exceptions. Therefore we consider the hypervisor framework as one contribution of this paper.

6 Implementation

This section provides the implementation details and discusses the issues that we encountered during the development.

Page Faults. Exception handling is the primary method of SMAPRO. We leverage SMAP exception to notify us when the kernel accesses userspace and use privilege-violation exceptions to solve the subsequent read and write conflicts. Those exceptions are all handled through the page fault handler, vector 14, in the IDT. The page fault is not an error, but an exception raised when accessing a virtual page. It is the MMU’s mechanism to implement virtual memory; thus, the page fault is recoverable.

We handle page fault exceptions before the kernel because, by design, the SMAP exception indicates a fatal error that the system should stop working immediately. We

also must be cautious about sending all the mitigation-irrelevant exceptions to the kernel. If we miss one, it may lead to system malfunctioning or crash processes.

Figure 4.11 shows a 32-bit error code that indicates the cause of the page fault. It is part of the trap frame, and the processor automatically pushes it into the kernel stack. However, there is no exact error code regarding SMAP. We make the cause conclusion from the trap frame and the values of CR2 and CR3.

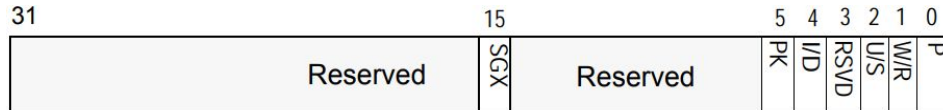


Figure 4.11: Page Fault Error Code. [68] **P: 0** The fault was caused by a non-present page; **1** the fault was caused by a page-level protection violation. **W/R: 0** The access causing the fault was a read; **1** the fault causing the fault was a write. **U/S: 0** A supervisor-mode access caused the fault; **1** A user-mode access caused the fault. **Notice**, there is no exact error code regarding SMAP. In the context of an SMAP exception, the U/S bit is zero, which indicates kernel-mode access. We still need to combine this information with the faulting address and CS segment register to confirm the cause.

We first filter out the mitigation-irrelevant exceptions because they may be nested exceptions caused by our operations, such as page table walking. For example, P:0 in the error code indicates a page absence, and we must pass it to the original page fault handler without any modification. The error code may have multiple bits combined, but we have not observed a SMAP exception on an invalid page so far. The U/S bit and CPL field in the CS segment register reveal whether the kernel caused the exception. Additionally, we keep track of the protected and relevant pages, which helps solve the subsequent conflicts. Algorithm 1 shows the basic algorithm to handle the exception.

Enabling Interrupt in the Page Fault Handler. The processor calls the page fault exception through an interrupt gate. Therefore when entering the page fault handler, the processor clears EFLAGS: IF bit automatically. It prevents subsequent interruptions from interfering with the handler’s execution, which we described in Section 2. In the page fault handler, we first store a copy of the CR2 register. Then set the IF flag before we walk the page table because it may contain swapped-out pages, and we need the interrupt enabled so that system can get noticed, thus bring them back

Algorithm 1 Page Fault Handler

```

1: procedure PAGEFAULTHANDLER
2:    $address \leftarrow cr2$ 
3:    $pte \leftarrow \mathbf{GetPte}(address)$ 
4:    $teb \leftarrow fs : 0x18$ 
5:   if SmapViolation then
6:      $pages[] \leftarrow \mathbf{AddPage}(address, pte, cr3, teb)$ 
7:     SetPageKernel(pte)
8:     FlashTlb(address)
9:     return Re-execute
10:  else if UserAccessProtectedPage then
11:    if error.WRITE then
12:      repeat
13:        Sleep
14:        if CheckPtePermits(pte) then
15:          return Re-execute
16:        end if
17:      until  $count < threshold$ 
18:      return TerminateThread
19:    else
20:      SetPgeUserReadonly(pte)
21:      FlashTlb(address)
22:      return Re-execute
23:    end if
24:  end if
25:  return OriginalHandler
26: end procedure

```

from the disk.

Hypervisor. Our light-weight hypervisor set the current system into a guest virtual machine. Unlike a bare-metal hypervisor, it does not emulate hardware devices. All the cores on the physical processor enter the VMX mode, and each core has its VMCS that represents a virtual core in the guest.

The virtual machine exits to the hypervisor when operating on control registers. We only focus on CR3 to monitor the process context switch. When the target process is about to run, the hypervisor sets CR4.SMAP bit in the Guest-state area of VMCS so that after re-entering the virtual machine, the SMAP is active on this core. Similarly, the hypervisor disables SMAP if the target process is switching out.

Updating Page Table. Any module that wants to update a page table should first hold the global spinlock to synchronize with other system components. However, as a third-party module, we do not have such conditions because the Windows kernel does not export this global lock. In a typical kernel-level TOCTOU attacking scenario, the

situation worsens because the attacker races with the kernel at high speed; in response, our mitigation also fastly operates on page table. Therefore, our code should be as atomic as possible to avoid conflicts with the kernel. Figure 4.12 shows the PTE data structure defined in C code. ① changes the Owner a.k.a U/S bit to zero, which sets this page to kernel-mode, and ② is the assembly code generated by a compiler. The one-line C statement becomes three assembly instructions. The problem is that, during the three instructions, the thread may be interrupted and switched out, and the value held in ECX may become outdated. It occurs many times when we test exploits against SMAPRO, making the mitigation ineffective. To solve this, we use the locked atomic instruction [69], as shown at ③.

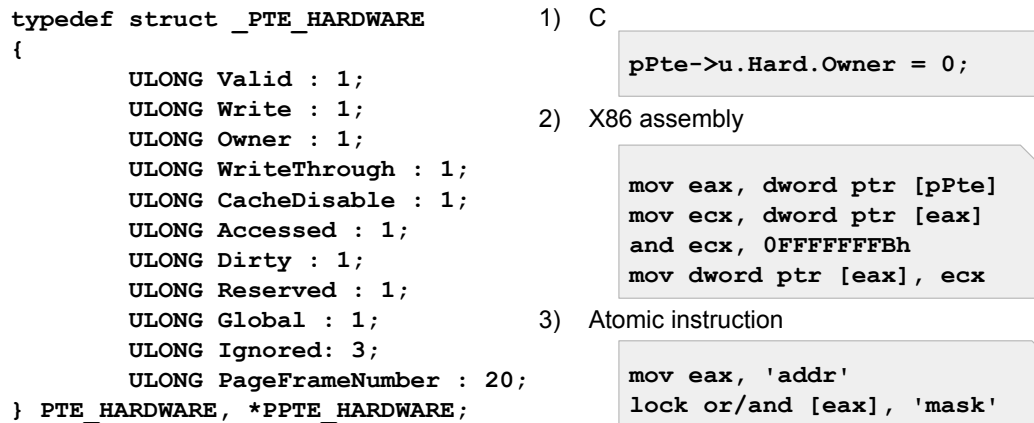


Figure 4.12: Left: PTE structure defined in C. Right: C code for changing one bit in the PTE structure, the corresponding assembly code generated by Microsoft C/C++ compiler, and the atomic instructions that serve the same purpose.

TLB Flushing. For SMAPRO, there is a difference between a real hardware computer and a virtual machine such as VMWare and QEMU. When we applied SMAPRO on the real hardware, it becomes ineffective because we did not flush the TLB. TLB is a memory cache that stores the recent translation of virtual memory to physical memory. The operating system needs to invalid the corresponding TLB entries when updates a page table. Instruction INVLPG only invalidates a TLB entry on the current processor. On a multi-processor system, each core has its TLB. Therefore, we need to issue inter-processor interruption (IPI) through local APIC to flush TLB on every core.

We find a Windows kernel internal function, `KeFlushSingleTb()`. It sends IPI

to all processors to invalidate a TLB entry. It calls the Windows hardware abstraction layer (HAL) to send the IPI, so the kernel does not need to mind the actual hardware differences. With this, our mitigation effectively works on the real hardware machine.

Intercepting System Calls. As mentioned in Section 5, to solve read conflicts, SMAPRO changes the protected page back to user-mode with a read-only permit. However, since it is no longer a kernel page, the attack can change its permits by calling Win32 APIs. Therefore, we need to intercept system calls such as `NtProtectVirtualMemory()` to prevent it. Nevertheless, there may be other ways to go around the system call protection, and we should improve it accordingly. It becomes another arms race between the attacker and the defender, which is out of this paper’s scope.

Terminating a Thread. When solving the write conflicts, the page fault handler suspends the thread and check the protected page’s status periodically. We set a retry threshold to avoid deadlock, and after a few attempts, we have to terminate the thread. However, terminating a thread is not a trivial task, so we use a trick to let the kernel handle it. We clear the P(present) bit in the PTE and the error code in the kernel stack. Thus, it looks like a thread reads invalid memory; then, we pass it to the kernel.

Special Cases. Windows is a complex operating system. There are always cases that we can not fully comprehend and need to handle specially. For example, `USER_SHARED_DATA` is a shared page between the kernel and user. Windows maps it at both `0x7FFE0000` and `0xFFDF0000`. The kernel frequently reads it because it contains settings such as system time. It is a read-only page, and we treat it differently to improve performance.

7 Evaluation

This section, we evaluate SMAPRO’s performance overhead and how it protects the operating system from real-world vulnerabilities.

7.1 Performance

In this section, we focus on performance evaluation. As previously mentioned, SMAPRO has two key components, the system module, and the hypervisor. To present the performance impact introduced by this mitigation, we conduct the tests in three parts.

All tests run on the PC with Intel Core i5-6400 (6th Gen CPU Skylake), ASUS H110M-C motherboard (Intel H110 Chipset, Realtek RTL8111H Network Controller), 8GB RAM, and 500GB hard disk.

Benchmarks. The hypervisor plays an essential part in SMAPRO. Without the hypervisor confining the SMAP feature, it would not be possible to debug a complex system like Windows with a system-wide unsupported feature. Nowadays, a hypervisor is part of the cloud computing infrastructure and can be regarded as a built-in component. The Windows 10 operating system even brings its native hypervisor to deliver security goals. Under those circumstances, SMAPRO can be added to the existing hypervisor with less performance overhead imposed on the system.

To evaluate the hypervisor’s performance overhead, we use the well-known benchmark SPEC 2006. We understand that this benchmark is for processor instruction set evaluation, specifically for microarchitectural aspects, such as instruction execution, branch prediction accuracy, cache policies. We choose several programs from the set. They are all non-GUI and computational intensive programs. Therefore the performance overhead incurred is primarily due to the hypervisor. Although our hypervisor and Windows HVCI have different objectives, we compare them to show that run-time protection utilizes virtualization techniques is practical.

Figure 4.13 shows that hypervisor’s performance overhead is acceptable, on average, 3.25%. HVCI yields a modest performance overhead of 0.81%.

Our hypervisor is slower than HVCI, particularly in two benchmark programs. Learning more about computer architecture and virtualization techniques, we wish to improve our hypervisor to perform better.

Non-trivial Applications. We also evaluate SMAPRO on several non-trivial applications, as shown in Figure 4.14. The applications we choose are meaningful

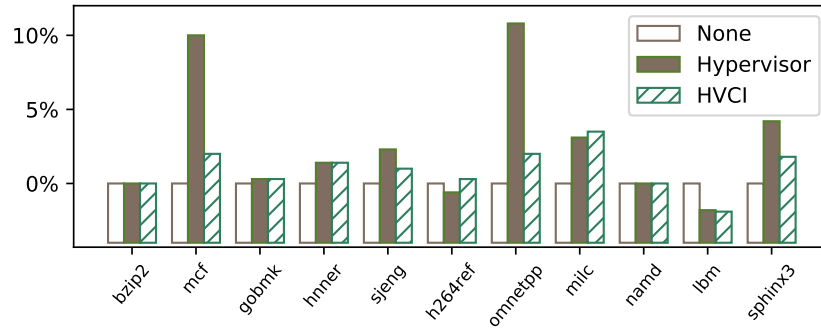


Figure 4.13: Performance overhead on the SPEC benchmarks incurred by the run-time load hypervisor. HVCi represent the Windows 10 native hypervisor for Hypervisor-Protected Code Integrity. All overheads are normalized to the unprotected system running benchmark.

because the kernel-TOCTOU vulnerability may threaten them in real-world scenarios. For example, a web server such as Nginx normally runs in a non-root account and takes external requests.

To test web servers, we count their response time for a web page request. For compression software, the test is to compress large files. We use speedtest1.c, which is a performance testing program for sqlite3. The result shows that the performance overhead is acceptable.

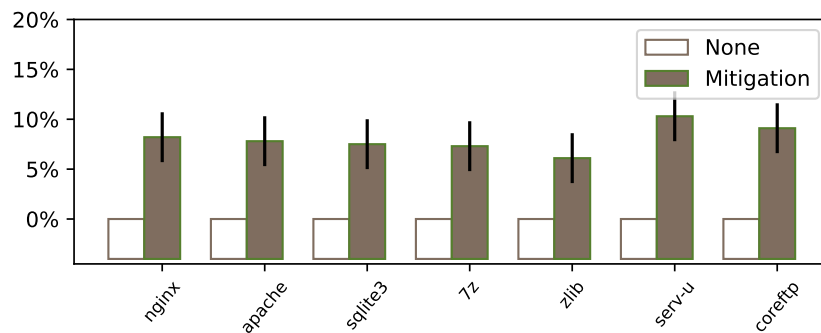


Figure 4.14: Performance overhead in non-trivial applications. Overhead mostly being introduced on system calls that need to fetch user parameters

GUI. Through our investigation, we find that the Windows graphical subsystem, namely, Win32k.sys, has the most double-fetch issues. Since GUI programs need to redraw their graphical components, they invoke the win32k system calls in a high frequency, even for a simple program such as Windows notepad. Therefore, our mitigation

incurs an unneglectable performance over GUI programs. The overhead is primarily affected by how often the interface refreshes. For example, if minimizing a GUI program's window, its performance will not be slow down by the graphical interface at all.

We compare GUI programs with non-GUI programs in the following aspects: the number of user pages accessed by kernel per system call and how many double fetches occur. We drag the GUI program's window to trigger redraw, and we send one URL request to the web servers per second. The measurement takes the first 500 system calls.

Table 4.2: System call count and user-pages accessed for GUI & non-GUI programs

Programs	System Calls	Protected Pages(r, w)	avg.	Double Fetch	Time (ms)
nginx	500	711(711, 0)	1.42	223	12312
apache	500	689(689, 0)	1.38	205	11339
notepad	500	1434(1102, 241)	2.87	1373	1859
freecell	500	1352(1165, 187)	2.70	1266	1500

Table 4.2 shows some interesting results. Refreshing the GUI takes tremendous system calls. As expected, both the kernel and the Win32k subsystem accesses user pages, but the win32k accesses more than the kernel does. We find that they both *capture* the user parameters at the beginning of each system call through reverse engineering, but the win32k module read/write user data even in the middle of a system call.

We also count the number of reads and writes on user pages. For non-GUI programs, the number of writes is zero, which is strange because most system calls need to write results back to the user program. With investigation, the causes are as follows. Windows provides three methods to transfer data between system calls and user programs, namely, Buffered I/O, Direct I/O, and Neither Buffered Nor Direct I/O. Among the three, the kernel mostly uses Buffer I/O and Direct I/O, which do not need to write to user-mode buffer directly. However, the kernel still needs to write user-mode variables such as the filehandle in system call `NtCreateFile()`. Figure 4.15 shows the pseudo-code similar to what the kernel uses to validate user parameters. We can see that the code always reads the variable first and the SMAP exception only captures first access. Therefore, this coding style is another cause for the zero writes. The result

shows that the kernel is well regulated on accessing user data. On the other side, the Win32k module has many writes, which tell a different story.

```

try {
    ...
    *((volatile HANDLE *)Address) = *Address;
} except(EXCEPTION_EXECUTE_HANDLER) {
    // Error handling
    ...
    return GetExceptionCode();
}

```

Figure 4.15: Pseudo-code for validating a file handler.

The column *Double Fetch* lists the user addresses that are read more than once during individual system calls. We trace this information with the TTFUZE as previously introduced in Section 4. Many of those double-fetch records are duplicates and benign. For example, the kernel needs to read data from Process Environment Block (PEB) or Thread Environment Block (TEB), or data structures located in userspace. However, the Win32k module has many more cases. Table 4.3 gives a glimpse of them. Every two rows indicate a double-fetch case, where after the first read, the subsequent instruction revisits the same address shortly after, and the identical CR3 and TEB show that two fetches are from the same thread of the same process.

Furthermore, we find that the Win32k module directly read the user variable not within a try-catch block in some of those cases. It is dangerous. The user programs can free the user memory and cause the kernel code to access an invalid page without protection, which leads to a kernel crash.

Table 4.3: In the selected double-fetch results, for every two lines, they have the same CR3 and TEB, which indicate the two records are from the same process, same thread. The two EIP are shortly apart, but the addresses they reference are the same user-mode address.

Cr3	Eip	Addr.	Teb
0x6d40320	0xbf812de4	0x4808c4	0x7ffdd000
0x6d40320	0xbf812e4b	0x4808c4	0x7ffdd000
0x6d40320	0xbf812dea	0x4808c8	0x7ffdd000
0x6d40320	0xbf812e55	0x4808c8	0x7ffdd000
0x6d40320	0xbf812daf	0x480750	0x7ffdd000
0x6d40320	0xbf812e21	0x480750	0x7ffdd000
0x6d40320	0xbf80c04d	0x7ffdd206	0x7ffdd000
0x6d40320	0xbf812ebe	0x7ffdd206	0x7ffdd000

To promote the performance of SMAPRO, it would be helpful if the unnecessary SMAP exceptions during parameter validating can be eliminated. As was done in the Linux kernel, the SMAP feature is temporarily disabled during `copy_from_user()` and `copy_to_user()`, the gateway functions. In Windows kernel, `ProbeForRead()` and `ProbeForWrite()` are the primary cause of SMAP exception. However, such *probe* has many variants such as `ProbeAndWriteHandle()`, `ProbeForWriteIoStatus()` and they are only partial of the user-data-copying code. Some of them are macros instead of functions, making it difficult to fix them without recompiling the kernel.

7.2 Case Study

CVE-2008-2252 is a kernel-level TOCTOU vulnerability reported by Thomas Garnier in 2008 and patched in Microsoft security bulletins ms08-061. It has been analyzed by many research works [162] [78]. To evaluate the effectiveness of SMAPRO, we test it on a real hardware machine. Since SMAP and SMEP are only available on a relatively new processor, we have to install an old system on a modern PC.

We write a program to exploit the CVE-2008-2252 vulnerability. To create a buffer overflow in the kernel, we need to enlarge the user-mode variable between the two kernel reads. The exploit creates two threads. Thread zero first allocates a virtual page at address zero for the parameters, which is necessary to bypass the sanity check of the system call `NtUserMessageCall()`. Then, it repeatedly calls the upper layer Win32 API `SendMessage()` (`WM_COPYDATA`) with the malicious parameters to open the attack time window. `SendMessage()` calls a lower layer function `NtUserMessageCall()`, which eventually calls the vulnerable win32k internal function `xxxInterSendMsgEx()`. Simultaneously, thread one keeps flipping the high bit of the target variable on page zero.

As shown in Figure 4.16, the kernel reads the user-mode variable (address 0x4) at ② and ③. Between the two instructions is the attack window within which thread one tries to enlarge the variable. To successfully mitigate the attack, we need to ensure this page remains unchanged during the time. The protection starts at ①, several instructions before ②, where the `CMP` instruction first reads the page hence raise an

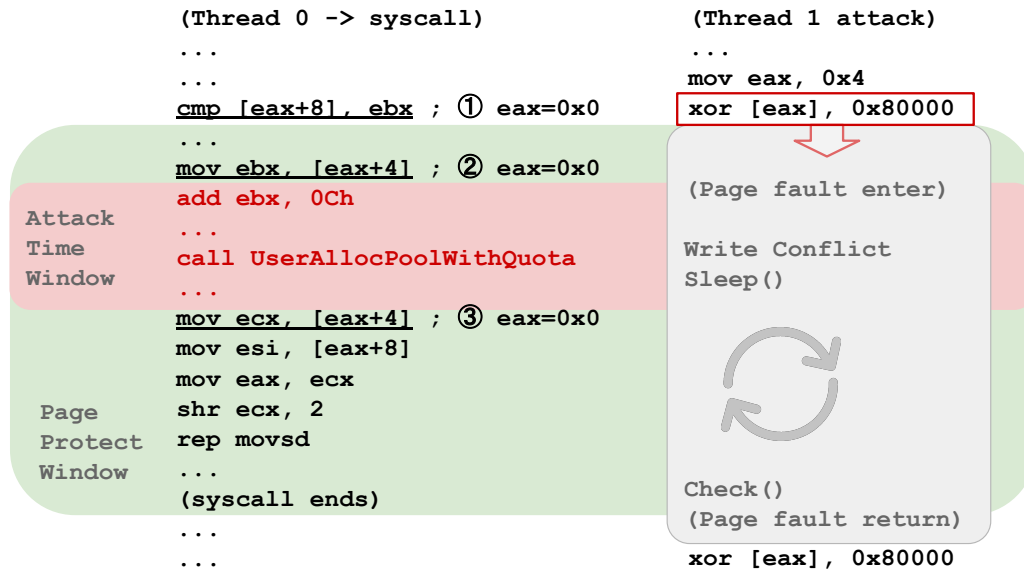


Figure 4.16: The attack thread tries to flip the data within the attack window, between instruction ② and ③. The kernel first touches the page at instruction ①, then the mitigation protects it afterward until the end of the current system call, creating a larger page-protect-window. As soon as the attack thread writes the protected page, the page fault handler suspends it until the protection ends.

SMAP exception. The protection continuously effective until the current system call `NtUserMessageCall()` ends. It covers the entire attack window. When thread one tries to tamper with the variable, it inevitably triggers a page fault regarding privilege violation. The page fault handler then suspends thread one until the system call ends, making it miss the opportunity.

CVE-2013-1254. Similar to the vulnerability above mentioned, CVE-2013-1254 [40] is another typical TOCTOU vulnerability. It is a family of 27 distinct vulnerabilities [107] [78] in the win32k module. It affects a variety of operating systems from Windows XP to Windows Server 2012.

These vulnerabilities share a similar pattern. The code is part of the parameters security check, around `_W32UserProbeAddress`, which is the highest possible address for user-mode data. Figure 4.17 shows that the flawed code first compares it with the passed in parameters' address, which is `ecx+8`. Only if the address is legit will the kernel uses it. However, after passing the security check, the kernel mistakenly reread the address. The attacker can abuse this, first pass the security check, then replace


```

.text:BF8A993F  mov     eax, _W32UserProbeAddr
               ...
.text:BF8A9973  cmp     [ecx+8], eax
.text:BF8A9976  jnb     short loc_BF8A997B
.text:BF8A9978  mov     eax, [ecx+8]
.text:BF8A997B
.text:BF8A997B loc_BF8A997B:
.text:BF8A997B  mov     ecx, [eax]
.text:BF8A997D  mov     eax, [eax+4]

```

Figure 4.17: The family of the 27 vulnerabilities in the win32k module is mostly around the parameter security check. The checking code with `_W32UserProbeAddress` makes sure that the parameter is from userspace. It is a classic time-of-check-to-time-of-use. When checking, the value is benign where `[ecx+8]` is less than `_W32UserProbeAddress`. After that, the attack can replace it with a malicious one before the second read.

the address with a malicious one, such as a kernel-mode address, which may lead to an arbitrary kernel write vulnerability depending on the case.

The protection takes effect when the instruction `cmp [ecx+8], eax` triggers the SMAP exception. The page is protected, and the kernel gets the same value at the second read.

8 Discussion

Fuzzing. Section 4 presents new method of fuzzing kernel TOCTOU vulnerabilities. A practical alternative is to utilize hardware data breakpoints. Data breakpoint [85] a.k.a watchpoint is a debug feature. It raises debug exceptions when accessing the set memory locations. DataCollider [85] use it to dynamically detects data races in kernel modules. Through static analysis, DataCollider first decides the sampling set. Then it inserts code breakpoints, and when one code breakpoint fires, it uses a data breakpoint to trap the second access to detect conflicts. Although only four data breakpoints are available on each processor core, it is sufficient for hunting data race bugs.

Qemu [15] with the dynamic binary translation (DBT) [47] engine is a unique approach for fuzzing. It fully perceives the details of every instruction emulated, and it is fast enough to run comprehensive tests on the latest operating system. Its dynamic translation backend is called the tiny code generator (TCG) [16] that converts binary code from the guest processor architecture to the host architecture. For example, it

can take x86 guest code and turn it into an equivalent sequence of instructions natively executable on an ARM host. It also can add instrumentation code [131]. QEMU with TCG is still slower than the hardware assistant virtualization, but it runs much faster than Bochs, making it capable of more in-depth fuzzing.

Write Conflicts. As previously mentioned in Section 5, we solve the write conflicts by making the thread suspending inside the page fault handler until the current system call ends. We also considered other methods to solve this issue, such as a thread-level copy-on-write (COW) mechanism. When there is a write conflict, the page table splits, and the protected page has two different mappings in each so that the kernel and the user thread have their copy. Therefore, the user thread can freely write it without affects the kernel. The hypervisor maintains the newly forked page table and only use it to replace the CR3 when this particular thread is running. We use similar techniques for controlling the thread context switching [126]. Other than the CR3 operation, we can monitor certain kernel data structures instead. However, the issue lies in determining the time to merge the page tables and deciding which copy to take when there are data conflicts. Because even at the end of the current system call, many places on the page may change. Without the precise timing sequence information, it is hard to decide whether they would cause kernel-level TOCTOU issues and which copy has the latest data.

Potential Attacks. When we discuss our mitigation with other security researchers, one researcher brought up a scenario as follows. Our mitigation protects a user page after the kernel access it. Although a user thread can not modify the page afterward, it can manipulate the kernel to do that. We acknowledged that this could be a possible attack because there are kernel vulnerabilities that give attacker kernel write capability. However, if the attacker already has such capabilities, the need to trigger another kernel-level TOCTOU vulnerability is questionable on a case-by-case basis.

9 Related Work

Transactional Memory. As discussed in Section 2, the root cause of kernel-level TOCTOU is that both kernel and user access the same memory address, which produce data inconsistency. Transactional memory [148] [132] [61] is a mechanism that allows a series of memory operations to execute atomically. It is an intuitive method for solving data inconsistency problems. Hardware transactional memory [58] [75] [35] [45] has become available on Intel processors since the Haswell macroarchitecture [132]. However, even before the hardware transactional memory feature is widely available, researchers use software transactional memory [81] [1] [56] to detect race conditions. It is more of a static analysis based approach.

With the hardware support, TxRace [176] detects data race at run-time. It instruments a multithreaded program to transform synchronization-free regions into transactions and leverage the conflict detection mechanism of Intel Restricted Transactional Memory (RTM). However, the hardware’s limitation is also apparent. Intel RTM does not support arbitrarily long transactions, simply aborting any transactions exceeding the hardware buffer’s capacity for transactional states. It can only run a short range of code and does not support a wide variety of processor mechanisms such as system calls, interruptions, special instructions (CPUID, MMX), IPI. Any of those and even access invalid memory may cause a transactional abort. Therefore, the hardware transactional memory is not suitable as run-time mitigation for kernel-level TOCTOU but a race condition detector. Although the idea of bug hunting is quite comparable to ours, similarly, we mark a whole system call as a transaction. Once we protect a user page during the system call, other user accesses cause the conflicts.

Dynamic Binary Instrumentation. It is a common method for monitoring a program’s behavior. Tools such as Intel Pin [98], DynamoRIO [121] injects instrumentation stubs into the program. It is a technique that widely used in data race detectors [138] [125] [173] [22] [99] [53] [128]. However dynamic binary instrumentation is only available for user-mode programs and the high performance overhead is the biggest limitation.

Hardware data breakpoints. DataCollider [48] leverage software code breakpoints and hardware data breakpoints to efficiently detecting data races in kernel modules. It randomly samples a small percentage of memory accesses as candidates for data-race detection. First, it uses static analysis to decide the sampling set, which are the instruction locations that access memory. Then it inserts code breakpoints, and when one code breakpoint fires, it uses a data breakpoint to trap the second access to detect conflicts. However, data breakpoint is a limited resource on commercial processors. An Intel processor usually has four hardware data breakpoints. Thus, it can only monitor a few program locations simultaneously. It is sufficient for hunting data race bugs, but not enough hardware resources for run-time kernel-level TOCTOU mitigation.

10 Conclusion

Kernel-level TOCTOU vulnerability is severe and widespread among operating system kernels. We learn its memory access pattern by studying real-world samples and develop a fuzzing tool TTFUZE that effectively finds kernel-level TOCTOU candidates in Windows.

Moreover, we creatively use an Intel hardware feature SMAP to mitigate such vulnerability. To the best of our knowledge, it is the first run-time protection for kernel-level TOCTOU. We test it against real-world vulnerabilities, and it effectively protects the Windows kernel from attacks. We also develop a lightweight hypervisor to confine the system-wide feature SMAP into specific processes. We evaluate the mitigation and lightweight hypervisor with 18 benchmark programs and real-world applications. The performance overhead is acceptable (less than 10% on average).

Chapter 5

Targeted Attacks against Cyber-Physical Infrastructures via Distributed Hardware Implants

1 Introduction

Critical infrastructure such as power grids comprises physical and cyber systems and assets that vital to national security. Their failure or incapacity would cause a significant impact on people's daily life on a large scale. Since the infrastructures systems are automated and computer-controlled, the industry informatization also brings security concerns.

The industrial control system (ICS) interconnects and controls the physical production assets. Compared to traditional IT infrastructures, the physical assets and the computer-based network's interconnection is a unique ICS feature. It is managed through an embedded system known as the programmable logic controller (PLC). In recent years, several worldwide incidents, such as Stuxnet [88], BlackEnergy [32] targeted critical infrastructures [26] [150] [177] [167] operated by ICS. Moreover, to sabotaging physical facilities, PLC is the preferred target of attack.

With the continuous emergence of attacks, protection measures have also been strengthened. ICS security has been traditionally handled using network security practices such as access control [49]. A common strategy is to use an isolated network from the Internet. Through physical access control, less software service is exposed to the public, reducing the potential attack surface, especially vulnerability-exploit-based attacks.

However, real-world APT attacks show that even an air-gapped network is penetratable [88]. The core of an APT attack is essentially a trojan backdoor that gains access

to a computer network and remains undetected for an extended period. The most crucial feature of a trojan is stealthiness. How to stay within the device is an essential issue that sophisticated APT attacks should consider. In particular, firmware modification [55] [123] [14] [21] [39] [84] [140] is the currently practiced attack plan. It injects malicious code into the target PLC, changes the working logic that runs in the device. However, this attack is subject to the firmware verification [105] [163] [93] [96] [145] [95] and update authentication [91] [113] [33] method. It would be much harder for the attacker to implant the malicious code once the firmware update is encrypted and digitally-signed and the system applies the methods mentioned above.

Another critical point for the APT attack is choosing the trigger event. Usually, the PLC has a dedicated real-time microcontroller to control the physical world through its IO pins. However, the microcontroller does not directly communicate with the host, the central control terminal (human-machine interface, HMI). Therefore, firmware modification attacks can perform a preset task individually, but it is difficult to react to PLC firmware updates or coordinate a distributed attack with other controlled nodes, especially among air-gapped networks.

We propose an alternative approach to circumvent existing software mitigations, HARDDOOR, a parasitical hardware implant inside a PLC attaching to its circuit board and remotely controlled through GSM network. With the recent emerging concept of supply chain attacks and real-world incidents [4], such a hardware attack appears to be more practical. The hardware implant can be pre-installed during the PLC device's assembly line or even during the shipment. We design it to be flexible because the PLC will load operating logic only after being deployed, and it is plausible that the ICS updates PLC's operating logic frequently. The hardware implant is specialized for the device's circuit board, the microcontroller, and other chips. It controls the IO through the digital signal and bus-level protocol hijacking, independent of the PLC's firmware.

Memory bus and interconnect protocols such as SPI, I2C are all potential targets. Low-speed protocols are prevalent due to their simplicity. For instance, JTAG (Joint Test Action Group) is an industry-standard for verifying designs and testing integrated circuits (IC) after manufacture. On ARM microcontrollers, extensive hardware features

are also provided through this interface for system-level debugging and tracing. It can read/write registers of processor and memory during system runtime. We leverage this interface for IO controlling purposes, and also, we can fetch the PLC’s firmware and operating logic for further offline analysis. Furthermore, with the ability to communicate through a GSM network, it is practical to control multiple nodes and organize a distributed attack simultaneously.

Contributions. To summarize, we make the following contributions in this paper:

- We present a novel attack class on industrial control systems: a parasitical hardware implant, which is completely invisible to the ICS control system.
- We disassembled and reverse engineered the circuit boards of a widely deployed Allen Bradley 1769-L18ER-BBIB CompactLogix 5370 PLC.
- We develop a prototype implementation of HARDDOOR, which is a small size device installed inside the Allen Bradley PLC.
- We write a JTAG driver that runs bare-metally on a microcontroller with minimal resource usage.
- We test and evaluate HARDDOOR and conduct a synchronized attack with multiple controlled Allen Bradley PLCs.

Roadmap. The rest of this paper is organized as the following. In Section 2, we provide the necessary background on programmable logic controllers and JTAG protocol. Section 3 describes the objectives, adversary model and scope, challenges, and architecture of HARDDOOR. Section 4 describes how we reverse-engineered the Allen Bradley PLC and prototyped HARDDOOR with implementation details (Section 5) and evaluation (Section 6), respectively. Section 7 provides a review of related work in the area of embedded system firmware attacks and their mitigations. We also discuss our views on the hardware backdoor and the possible mitigation strategies in Section 8. Finally Section 9 concludes the paper.

2 Background

This section provides background knowledge for the rest of the paper. We first introduce the industrial control system (ICS) and provide detailed information about the programmable logic controller (PLC). Then we give a more detailed technical background on JTAG and I2C protocols, which are heavily used in this project.

ICS is a distributed system used for industrial process control. It connects sensors and actuators that interact with the physical systems (e.g., power grid) with the cyber components such as networks and servers. In a factory, local operations are often controlled by PLCs that receive supervisory commands from a remote host. For example, a human operator monitors the system's state and sends out instructions through a human-machine interface (HMI). Most PLCs and HMI hosts are connected to the ICS via Ethernet.

PLC. PLCs consist of a microcontroller, I/O modules, a power supply, and other specialized add-on modules. The I/O modules of PLCs interact with the physical world, gathering digital inputs from sensors, switch, or a thermometer. The microcontroller serves the PLC's brain, executing pre-programmed ladder logic based on the inputs and giving operation signals through the output module. Essentially, a PLC is an embedded system. They both use a microcontroller and have limited computing resources compared to modern computers. However, PLCs are specially designed and rigorously tested to withstand operating in an industrial environment where they may be exposed to vibration and noise.

The firmware on PLC contains either a real-time operating system or code that runs bare-metally on the microcontroller. The PLC usually also provides library routines to operate on various I/O devices such as I2C, SPI [94] and CAN bus [24]. In this project, the PLC we work with provides the library as ROM code, which we further discuss later.

The PLC usually uses a fix-interval timer to run compiled ladder logic repeatedly, which is the so-called scan cycle. Moreover, the way that PLC interacts with the IO module is through the GPIO ports on the microcontroller. Essentially, each digital

input/output in the PLC has a corresponding GPIO bit. The PLC also has an LED panel that indicates each IO pin's status, which roughly gives an idea of whether the PLC functions correctly.

JTAG is the common name of the IEEE1149.1 standard, which defines the Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards using boundary-scan. The JTAG interface uses very few pins (TDI, TDO, TMS, TCK, and TRST) to connect to an on-chip Test Access Port (TAP) that implements a stateful protocol, as shown in Figure 5.1. One or more devices can expose multiple TAPs in a daisy chain, also known as a scan chain. The host communicates with the TAPs by manipulating TMS and TDI in conjunction with TCK and reading results through TDO.

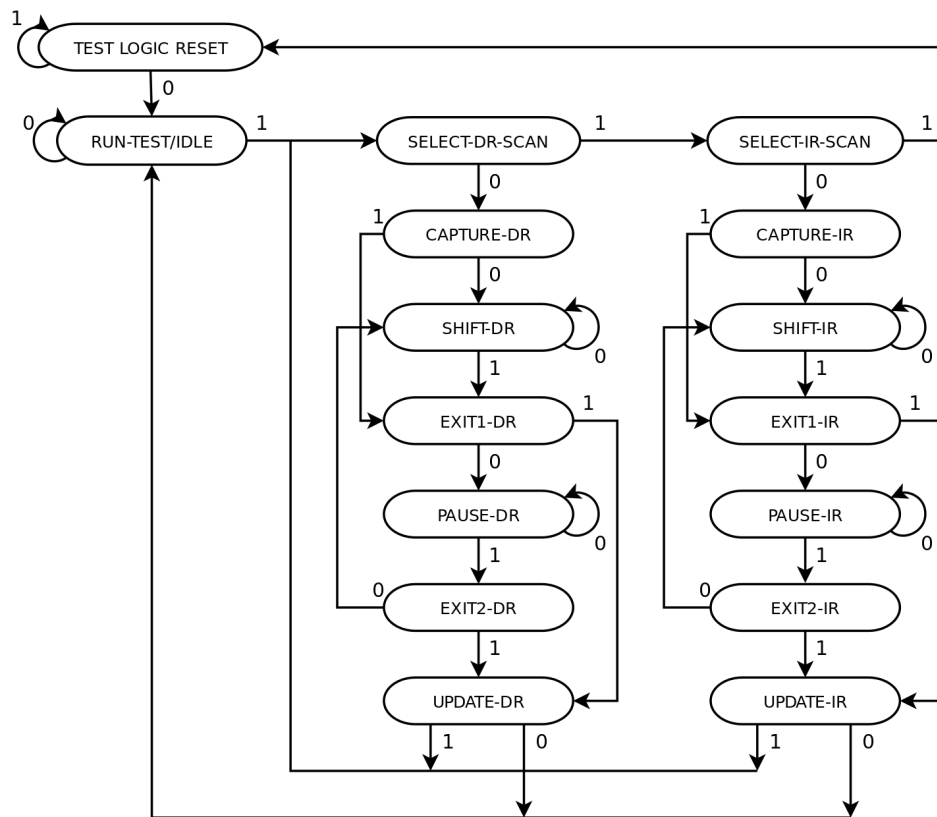


Figure 5.1: JTAG TAP state machine.

The JTAG standard has four common registers: Instruction Register (IR) and Data Register (DR), IDCODE, and BYPASS. The IDCODE register contains data that uses a standardized format that includes a manufacturer code. The BYPASS register is a

single-bit data register that allows this device to be bypassed (do nothing) while other devices in the scan chain are examined. The IR and DR register's size depends on the TAP implementation, and they are used to send in instruction and receive result data.

The TAP implementation defines instructions and associated them with internal data registers. For instance, the host sends the IDCODE instruction through IR and subsequently gets the value of a 32-bit register (IDCODE) from TDO.

The PLC we used in this project uses an ARM core microcontroller, namely, Texas Instruments Stellaris LM3S2793 [65]. The debug functionality provided in LM3S2793 is as CoreSight components. It provides real-time access for the debugger without halting the processor to AMBA (Advanced Microcontroller Bus Architecture) [54] system memory, peripheral registers, and all debug configuration registers. The TAP controller is implemented using CoreSight technologies, and it is called Debug Access Port (DAP) instead.

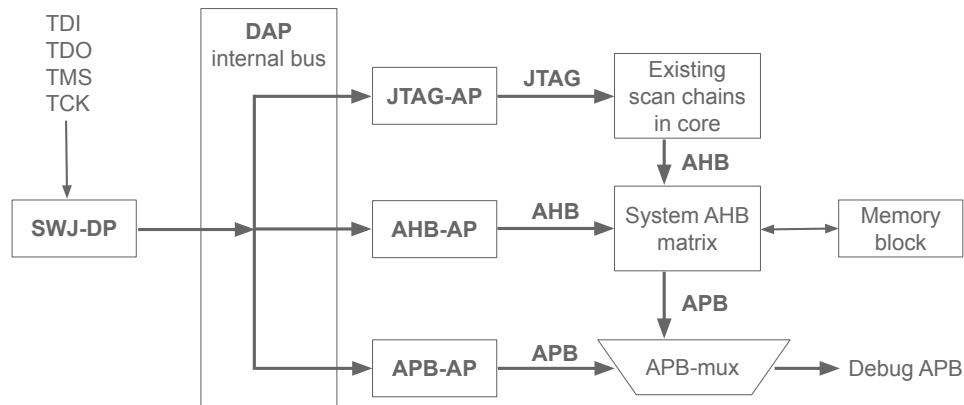


Figure 5.2: The debug port (DP) receives host signals and maintains the JTAG state machine. The instruction and data are sent to the DAP core through IR and DR, respectively. Unlike the JTAG standard that needs to halt the processor before reading registers, using CoreSight DAP, the registers, SARM, and MMIO can be accessed during runtime without halting the processor.

As shown in Figure 5.2, each DAP contains Debug Ports (DPs) and Access Ports (APs). The DP provides access to the DAP from an external debugger. Then the DAP uses the APs to access on-chip resources. Multiple APs such as AHP-AP, ABP-AP, and JTAG-AP respond to each type of bus and the devices that connect to it. For instance, the AHB-AP provides an AHB-Lite master for accessing the system AHB bus, which

we use to access the RAM and MMIO.

I2C is a serial protocol that connects low-speed devices in embedded systems. The I2C bus only has two wires, namely, SCL and SDA (the third wire connects to the ground). SCL is the clock line that synchronizes all data transfers over the I2C bus, and the SDA is the data line. All the low-speed devices in the system can be connected to the same I2C bus as slave devices, where each device has a unique address. The master device initiates a transaction by sending a high-to-low signal on the SDA while keeps SCL high. It is called a start condition. After that, the master sends the target device address byte onto the bus. The first 7 bits are the address, and the last bit indicates the data direction where one indicates reading, zero indicates writing. Only the device that matches this address continues with this transaction. It acknowledges this byte by pulling SDA low during the next SCL pulse. After addressing the target device, the master sends out the target device's internal location or register number. Next, the master device sends the data. The target device automatically increases its internal register address after receiving each byte. When the transaction is complete, the master device sends a stop sequence onto the bus.

3 Overview

Critical infrastructure such as power grids comprises physical and cyber systems and assets that are vital to national security. Their failure or incapacity would have a significant impact on people's daily life on a large scale. The recent Ukrainian power grid attack, the massive blackout in south American countries [57] demonstrated that influence not only affects people's livelihoods but even international politics. The cyberattack on the ICS system can even cause physical damage to the infrastructure [174], which makes it harder to recover. Incidents such as the Stuxnet proved this point.

Consequently, ICS has received considerable attention due to security concerns. There are many ways to breach a computer system, and most of them are focus on software-based approaches such as vulnerability hunting and exploiting, cracking of authentication and protocols. Therefore, the attacker needs to break into the network

and avoid existing access control and other mitigations. Moreover, most critical infrastructures use their air-gapped network, and It may take a state-sponsored team to accomplish such as mission [88].

Under such circumstances, we believe that cyber-attacks with the assistance of physical approaches are underestimated, especially before the emergence of the so-called supply-chain attacks. Among those world-class attacks, the term APT(Advanced Persistent Threat) is often mentioned. In essence, the APT attack is an extremely well-hidden trojan that can be deployed for many years without being detected. Thus, installing the trojan and remotely triggering it is the crucial point of a successful attack.

In this paper, we provide two plans for physically installing a trojan and provide a prototype hardware implant HARDDOOR that can be used for this type of attack. Figure 5.3 depict the scenario. In many parts of the supply chain, such as the factory and shipment, numerous employees have the opportunity to access the PLC. Installing an extra piece of a circuit board is not a difficult task for professionals. Moreover, a large-scale infrastructure such as a power grid has many remote substations with very few staff. An attacker can sneak into one of the substations and install the hardware backdoor. The difficulty between the targeted PLC and the attacker, in reality, can be just a few padlocks. We believe that the breach of a substation can cause a chain reaction in a power grid, and it is a real threat [127] [28].

After the attacker controls enough PLCs, he can remotely initiate a distributed attack to cause more significant damage using a cellular network.

The advantage of this attack is that it does not rely on the existing ICS network, nor is it limited to the firmware running on PLCs so that it can evade most software-based mitigations. It only needs to know the specific PLC model of the target and modify the hardware implant accordingly.

3.1 Adversary Model

The attacker knows the target’s specific PLC model and can obtain the exact model for studying.

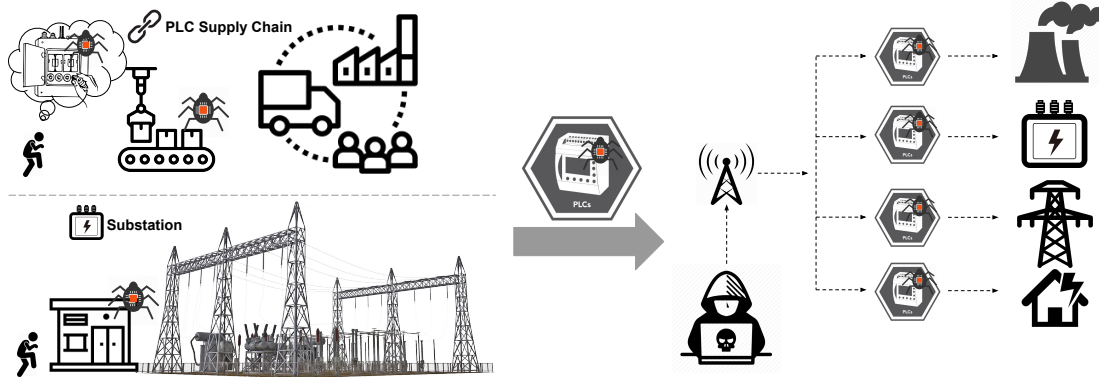


Figure 5.3: In the PLC manufacturer factory or during the product shipment, numerous employees can access the PLCs. The hardware backdoor installation should follow specific procedures to be efficiently accomplished without advanced software knowledge. The hardware backdoor can communicate with the attacker through the GSM network. Therefore it does not need to join the ethernet used by the ICS system.

The printed circuit board (PCB) and the IC chips of the PLC should be well exposed. For example, the Chip-on-Board (COB) [89] packing brings extra challenges for the attacker. The black glob-top makes it no easy to identify the chip model and pins. Fortunately, high-end microcontroller products rarely use this packaging. It would be a great advantage that the attacker can use the JTAG interface of the microcontroller. In other words, the JTAG interface is not disabled by programming fusing bits at the factory. Nevertheless, the attacker can control the IO or tamper with the firmware image when transferred through the bus without JTAG.

To remotely control the device, the attacker uses a GSM network or WIFI to communicate with the hardware backdoor. The PLC must not be deployed in an electromagnetic isolation environment where the wireless signal can not be transmitted outside.

The operating system and software mitigation that runs on the microcontroller does not affect the backdoor. Therefore the PLC system allows having any memory protection based on MMU [147] and MPU [83].

3.2 Challenges and Approaches

The major challenge in attacking PLCs is not having enough information about the device. Some vendors publish the microcontroller's datasheet, but some vendors use

proprietary design with highly customized instruction set architecture (ISA). The layout of the PCB board and the on-board pin definition are also not publicly available.

Firmware. In an embedded system, flash memory usually stores a file system and a real-time operating system (RTOS) such as VxWorks [122]. It is the so-called firmware. Specific to a real-time microcontroller, the firmware runs bare-metally on the microcontroller or with a lightweight RTOS such as FreeRTOS [13]. The PLC we use in this project is Allen Bradley 1769-L18ER-BBIB CompactLogix 5370. Through JTAG, we read the flash and ROM memory out of the microcontroller. Moreover, we also ported a driver to read an on-board SPI flash chip, namely, AT45DB021. The source code is available at the github ¹. We can identify the PLC use's internal data structures and libraries used with some reverse engineering effort.

JTAG Pins. On some boards, the JTAG pins are difficult to trace, mainly when the microcontroller chip uses BGA packaging [76] that all the balls are buried underneath. In our case, there is a 10-pin solder pad, as shown in Figure 5.4, which is likely to be an unsoldered JTAG socket (we also own one PLC that the socket is soldered on). To identify each pin, we use a multimeter to conduct the connectivity test. The microcontroller supports both JTAG and Serial Wire Debug (SWD)[11] interfaces.

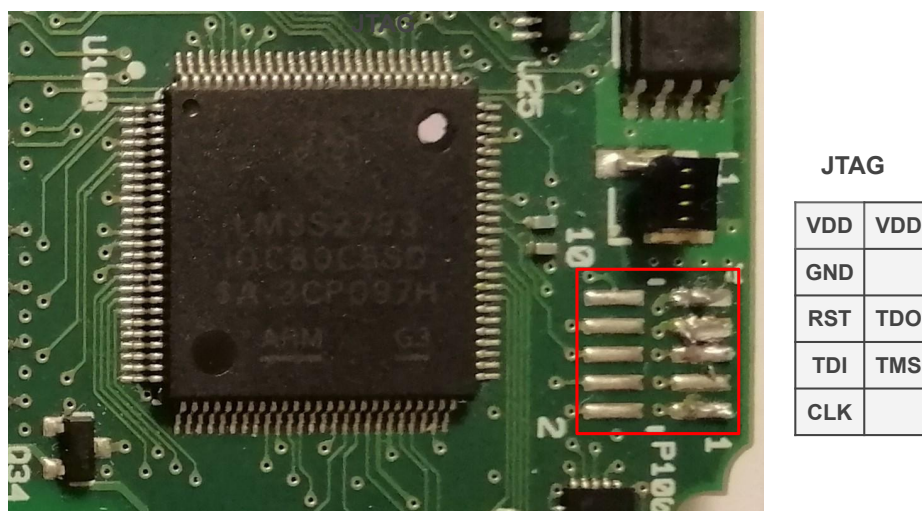


Figure 5.4: JTAG pins on the real-time microcontroller board. A square pad with ten pins is very likely to be a JTAG interface. We need at least to have the TDI, TDO, TMS, and CLK. RST is optional.

¹https://github.com/whensungoesdown/at45db021_teeny32

JTAG Protocol. To make our hardware backdoor small, we use the Teensy 3.2 development board. We also code a driver to send JTAG commands, hence accessing the microcontroller’s on-chip resource. It is the equivalent of a trivial hardware debugger. Although the open-sourced debugger OpenOCD [62] supports many platforms, the Cortex-M3 core that the Teensy has can not support its runtime environment. Our driver runs bare-metally on the microcontroller with minimal resource usage, and it can be ported to an even more restrained embedded system. We also consider it as one of the paper’s contributions. The source code is available at the github ².

Remote Control is a core function of the hardware backdoor. To avoid existing access control and even penetrate an air-gapped network, we use the SIM800C module to communicate with the backdoor through a separate GSM network. It connects with the Teensy board using a serial port.

4 Reverse Engineering and Design

The major challenge in making a PLC-specific hardware backdoor is that PLC does not use open standards like PC. During the development, a significant amount of time is spent on the reverse engineering of the PLC. After a basic understanding of its modules and chips, we can choose how to control it, whether through JTAG or intercepting other low-speed buses.

4.1 Reverse Engineering

The PLC we use is Allen-Bradley 1769-L18ER-BB1B/B CompactLogix 5370. The reverse engineering work includes both the hardware and software parts. First, we speculate on the function of each board of the PLC. Then we identify the on-board IC chips and use a multimeter to conduct the connectivity test for wire tracing. The purpose of this is to find the interconnections between chips and between boards. Fortunately, the real-time module is a two-side PCB, and the wire connectors that pass through the board are well exposed, as seen in Figure 5.9. Furthermore, we dump the

²https://github.com/whensungoesdown/teensy_jtag

firmware from the microcontroller and flash chip for reverse-engineering analysis.

Backplanes. The Allen Bradley PLC contains several PCB module boards, known as backplanes. Figure 5.5 shows each module. In this paper, we name the board with the Ethernet socket and USB port as the communication module (**B**). The one controls that has a microcontroller and controls the 16 digital DC input pins and 16 digital DC output pins as the real-time module **C**, which is our main target. Module **D** is merely a connector for the digital IO. The rest are the power supply module(**A**) and LED module(**E**).

The communication module (**B**) itself is an embedded system, including a CPU, DRAM, and other peripherals such as Ethernet, SD card, and USB. It communicates with the host (HMI) to receive firmware and ladder logic updates, and it also hosts a web server to display status. However, this module does not directly interact with IO. By analyzing the PLC firmware update files, we find that this board runs VxWorks operating system. Nevertheless, this board’s primary chip is an FPGA [86] chip that runs a soft-core ARM processor, and it is in BGA packaging. It is not easy to trace and identify which GPIO pins of the FPGA are used as the JTAG interface. We consider it as one of our further works. Therefore, our focus is the real-time module (**C**) that runs ladder logic and directly controls IO. Fortunately, this module uses a commercial microcontroller, namely, Texas Instruments Stellaris LM3S2793 SoC.

The digital IO sockets on the connector module (**D**) connects to the real-time module (**B**). The IO goes through the power switches and optically coupled isolator chips and eventually connects to the microcontroller, as shown in Figure 5.12. There are 32 LEDs corresponds to each IO socket, and the real-time module also controls the LED module through the I2C protocol. The power module (**A**) provides stable 3 volts for other boards. Our backdoor can either get power directly from it or the JTAG pad.

Microcontroller. The TI Stellaris LM3S2793 SoC has an ARM Cortex-M3 processor core that operates at 80 MHz. It contains 64 KB SRAM and 128 KB flash. The internal ROM is preprogrammed with Stellaris Peripheral Driver Library (DriverLib) to drive the on-chip peripheral devices. Table 5.1 shows the memory map.

Start	End	Description
Memory (0x00000000 - 0x22200000)		
0x00000000	0x0001FFFF	On-chip Flash
0x00020000	0x00FFFFFF	Reserved
0x01000000	0x01004FFF	On-chip ROM
0x01005000	0x01005EFF	AES+CRC lib in on-chip ROM
...		
0x20000000	0x2000FFFF	Bit-banded on-chip SRAM
...		
FiRM Peripherals (0x40000000 - 0x4001FFFF)		
...		
0x40008000	0x40008FFF	SSI0
...		
Peripherals (0x40020000 - 0xDFFFFFFF)		
0x40020000	0x400207FF	I2C Master 0
...		
0x4005C000	0x4005CFFF	GPIO Port E (AHB aperture)
0x4005D000	0x4005DFFF	GPIO Port F (AHB aperture)
0x4005E000	0x4005EFFF	GPIO Port G (AHB aperture)
0x4005F000	0x4005FFFF	GPIO Port H (AHB aperture)
...		
Private Peripheral Bus (0xE0000000 - 0xFFFFFFFF)		
...		
0xE000E000	0xE000EFFF	Nested Vectored Interrupt Controller (NVIC)
...		

Table 5.1: LM3S2793 Memory Map. Only list the address space of the memory and devices related to this paper. FiRM-compliant (compliant to the ARM Foundation IP for Real-Time Microcontrollers specification).

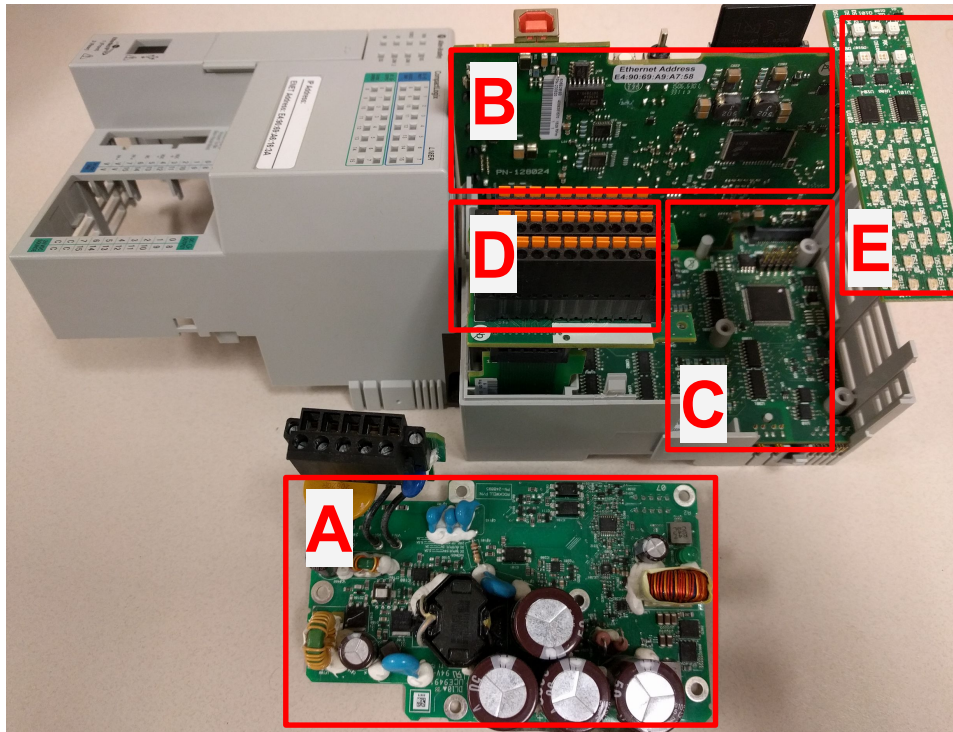


Figure 5.5: Allen-Bradley 1769-L18ER-BB1B/B CompactLogix 5370 PLC. A: Power supply module B: Communication module C: Real-time module D: (16) DC Digital Outputs & (16) DC Digital Inputs Connector E: LED module

Reset Vector. The vector table is at a fixed address 0x00000000 after the system reset. The core starts to execute from memory 0x00000004, which is the reset vector. Table 5.1 shows that the reset vector resides in the flash memory instead of the ROM. It is because the ROM boot loader is only executed in two scenarios. The first case is when the flash memory is empty. The other one is when an application initiates a firmware update and calls the ROM boot loader to execute. For instance, if data at 0x00000004 is 0xFFFFFFFF, which indicates an empty flash, then the ROM is mapped to 0x00000000 to substitute the flash and execute instead.

The data at 0x00000000 and 0x00000004 will be loaded into the stack pointer (SP) and the program counter (PC), respectively. In our case, the SP is 0x20000B48, and the PC is 0x000000E3. Notice that 0xE3 is an odd number. As we know, RISC processors such as ARM uses fixed-length instruction. On ARM processors, setting the PC's least significant bit indicates that the following code will be executed as the

ROM_APITABLE (0x1000010)
[0] = ROM_VERSION
[1] = pointer to ROM_UARTTABLE
[2] = pointer to ROM_SSITABLE
[3] = pointer to ROM_I2CTABLE
[4] = pointer to ROM_GPIOTABLE
[5] = pointer to ROM_ADCTABLE
...

Table 5.2: LM3S2793 ROM API table is at a fixed address 0x1000010, and each table entry is 4 bytes address that points to a second-level table, which corresponds to a class of peripheral devices.

two-byte Thumb instructions. Therefore, we dump the flash memory and disassemble it at address 0x000000E2 instead.

Flash Boot Loader. Right after the flash boot loader starts, it copies itself to SRAM, starting at 0x20000000. Although both SRAM and flash memory can be accessed in a single cycle, flash memory can do that as long as the code is linear and branches incur a one-cycle stall. The bootloader copies 0x00000000 - 0x00000A88 to 0x20000000 - 0x20000A88, and clear the data section (0x20000A88 - 0x20000F54). As mentioned earlier, on system reset, the vector table is at the fixed address 0x00000000. However, it can be relocated by writing the vector table offset register (VTOR:0xE000ED08). Changing the vector table indicates a complete change of the system's behavior because all the interrupt handlers are new, and they interpret how the system behaves regarding peripheral device's requests. The bootloader sets the vector table to 0x20000000 and jumps back to SRAM to continue execution.

Stellaris Peripheral Driver Library. The Drivelib [66] is a set of APIs utilized to control the on-chip peripheral devices. It is provided in ROM code and placed in a fixed location on Cortex-M3 SoCs. It helps identify what functions the firmware calls. Through the fixed location of APIs and the Drivelib datasheet, the function matches with names. It is a two-level table structure. The main table is at 0x1000010, and it contains the address of the second-level table for each type of peripherals, as shown in Table 5.2. For example, Table 5.3 shows the ROM_GPIOTABLE, which contains all the GPIO related APIs.

ROM_GPIOTABLE
[0] = function ROM_GPIOPinWrite
[1] = function ROM_GPIODirModeSet
[2] = function ROM_GPIODirModeGet
[3] = function ROM_GPIOIntTypeSet
[4] = function ROM_GPIOIntTypeGet
...

Table 5.3: GPIO API Table. Each table entry is the entry address of an API, and the function parameters are passed through registers. There is no privilege or mode change when calling into the APIs.

Figure 5.6 shows a typical code snippet that calls a ROM API. The second-level table is at $0x1000010 + 0x10$ (`ROM_APITABLE[4]`), which is the the GPIO table. And the API it calls is `ROM_GPIOPinRead()` (`ROM_GPIOTABLE[11]`). Because all the ROM API call sites have the same pattern, it makes firmware’s reverse engineering work much easier. Calling the API is merely locating the function address from the two-level tables and jumps to it, and the parameters are passed in through registers. There is no privilege change when calling into the APIs, and the firmware code all runs in the system mode.

```

MOV      R1, #1
LDR.W    R0, =0x40059000 ; GPIO Port B
LDR.W    R2, =0x1000020
LDR      R2, [R2]
LDR      R2, [R2, #0x2C]
BLX      R2           ; ROM_GPIOPinRead
                        ; Reads the values present
                        ; of the specified pin(s).

```

Figure 5.6: A code snippet that calls `ROM_GPIOPinRead()`. The parameters are passed through registers. In this case, the `ROM_GPIOPinRead()` has two parameters. R0 contains the GPIO port address, and R1 indicates the pins to operate.

GPIO. The PLC interacts with the physical world through digital inputs and outputs. The IO goes through the power switches and optically coupled isolator chips and eventually connects to the microcontroller’s GPIO.

In the microcontroller LM3S2793, the GPIO module comprises nine physical GPIO blocks, and each corresponds to an individual GPIO port. Depending on the microcontroller’s configuration, it supports up to 67 programmable input/output pins or several

pins grouped to provide peripheral functions such as I2C. The GPIO ports can be accessed either through AHB or APB bus, which matters when we access the GPIOs through JTAG. We choose the AHB-AP.

Each GPIO port has several associated control registers, such as GPIO Digital Enable (GPIODEN), GPIO Alternate Function Select (GPIOAFSEL), GPIO Port Control (GPIOPCTL), and GPIO Data Control registers. All GPIO pins are configured as individual input/output pins and tri-stated by default. The GPIO Direction (GPIODIR) register configures each GPIO pin as an input or output, and we do not change it. To change the inputs and outputs, we primarily operate on the GPIO Data (GPIODATA) register that modifies individual bits in GPIO ports.

The way to control the GPIO data is not straightforward. Different microcontrollers adopt different operating methods. For example, the GPIO port may take Output Data Register (ODR), Bit Reset Register (BRR), Bit Set/Reset Register (BSRR) [38]. In our case, the LM3S2793 microcontroller uses a more complicated model to conduct bit-wise operations. The GPIO Data register is memory-mapped. When read/write, bits[9:2] of the address are treated as a bitmask, as well as the bits[1:0] are always zero because the memory access should be at 4-byte alignment on ARM. Therefore, for each GPIO port, the memory-mapped range should be 1KB long, that is, from GPIODATA to GPIODATA + 0x3FC. A write can only change the data bit when the corresponded bitmask is set. Otherwise, the data bit is unchangeable.

For example, GPIO Port A (AHB) is mapped at 0x40058000, and it controls 8 GPIO pins. We want to set bit2 to 0 and bit5 to 1. As shown in Figure 5.7, the bitmask is 0x90, and the data is 0xF0. Hence, the operation should be writing 0xF0 to address 0x40058090.

The same applies to reading. Only the corresponding bit in the bitmask will be read. Otherwise, it reads zero. For example, to read the four high bits from GPIO Port A, the address with bitmask should be 0x40058000 + 0x3C0, and it reads 0xA0, as shown in Figure 5.8.

Control Output. After knowing how to control GPIO, we can directly control the output of the PLC. There are 16 inputs and 16 outputs on the IO connector.

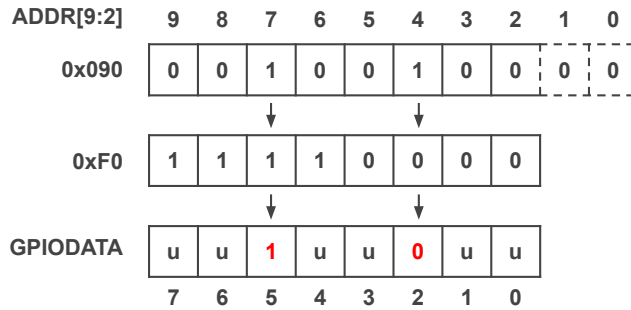


Figure 5.7: Writing a byte of 0xF0 to address GPIODATA + 0x90. The bitmask only allows bit2 and bit5 to be modified. Therefore, only two bits are valid for the write operation. **u** indicates the new bit is ignored.

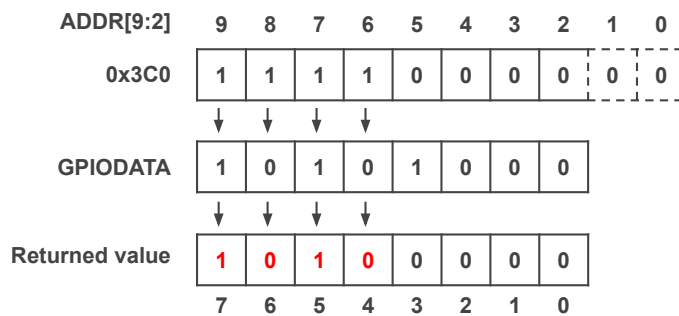


Figure 5.8: The address GPIODATA+0x3C0 reads the high four bits of the GPIO port. The rest reads zero regardless of the actual value.

Through reverse engineering, we know the GPIO port corresponding to the IO. That are, GPIO port E (0x4005C000) GPIO port F (0x4005D000) for inputs, and GPIO port G (0x4005E000), GPIO port H (0x4005F000) for outputs. Intuitively, each GPIO bit corresponds to a pin on the connector. One indicates the high voltage, which is the field power voltage; zero dictates the low voltage (8 volts).

To conduct a stealthy attack, we want to change the output secretly. For example, we want to keep the LEDs in their original state and the host (HMI) not to find any abnormalities. To achieve this goal, we need to leverage the firmware itself. The PLC periodically scans the inputs, runs ladder logic, and updates outputs. It is trivial to find the ladder logic binary by following the timer handler routine. There are a few local variables that control how the ladder logic behaves. For example, one local variable determines if any logic state has changed. If so, the corresponded GPIO pin will be updated according to the ladder logic. We modify this local variable so that everything looks up to date. In the meantime, we can change the outputs without triggering any

alarms.

AT45DB021E SPI Flash Memory. It is quite noticeable that right next to the LM3S2793 microcontroller, there is an 8-pin flash chip, which is an Adesto 45DB021E 2-Mbit SPI flash memory. It connects with the PLC's SSI0 (Synchronous Serial Interface), as shown in Figure 5.9.

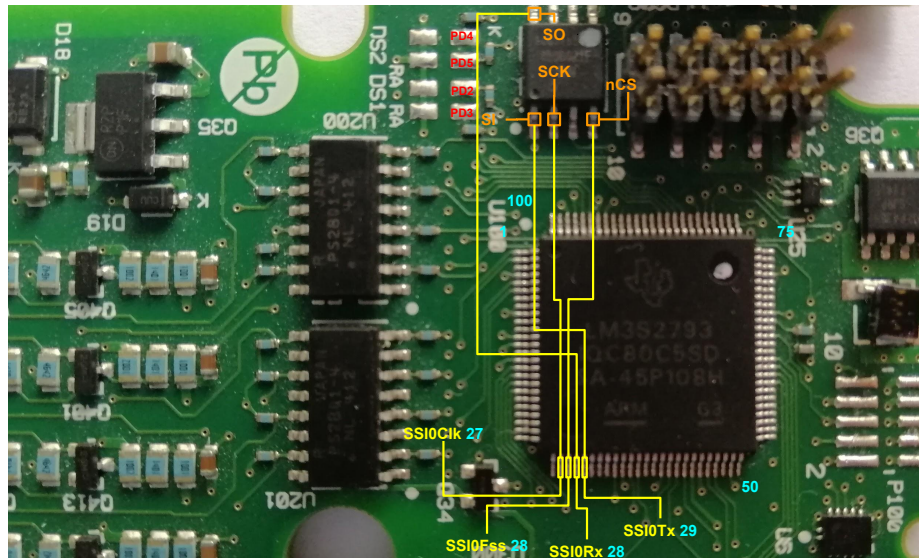


Figure 5.9: Through wiring tracing, we find that the AT45DB21E SPI Flash Memory connects to LM3S2793's SSI0 interface. The eight solder joints on the left may be used to install four LEDs. They are controlled by GPIO port D, but they are not installed.

During the boot process, the pins in GPIO port A are assigned for the SSI0 master device. The firmware first reads one byte from the flash chip (offset 0x2000), which looks like a status byte. If it equals 0x55 or 0xAA, the PLC will be reset. If not, the firmware checks the integrity of the address 0x4000 to 0x1FFFC, the compiled ladder logic. The algorithm is a simple checksum. Accumulate each byte in this address range, and the result should be equal to the byte in address 0x1FFFF.

So this status byte at 0x2000 indicates the status of the PLC last time it was running. The value 0x55 indicates that the system has encountered a severe failure. If so, the firmware operates on GPIO port D, leading to the eight solder joins next to the SPI flash. We think they may be four LEDs to show the status. Furthermore, if the value is 0xAA, it means that the ladder logic binary has been broken, and the firmware will copy the code from 0x6100 in the SPI flash. We think this is a backup code and

also a place where malicious code can be stored.

To read the AT45DB21E flash chip's content, we port its driver to the Teensy 3.2 board. The source code is available at the github ³.

Front Panel LED. There are four rows of LEDs on the front panel of the PLC. Each LED represents the state of an input or output. Usually, this is a very intuitive reflection of the current status of the PLC.

There are four rows of LED lights on the PLC's front panel. Each LED shows individual input and output pins' status, which is an intuitive way for the administrator to check whether the device works properly. The microcontroller controls these LEDs through the I2C protocol [141].

There are two 24-pin PD9535 chips (Remote 16-bit I2C/SMBus, Low-Power I/O Expander [67]) on module **E**. It provides general-purpose I/O expansion for microcontrollers via the I2C bus. The two pins in GPIO port B(PB2 and PB3) on the microcontroller are used as the SCL and SDA signal lines for the I2C master device. These two signal lines also pass through the connector between module **B** and **C**, as shown in Figure 5.13.

Each I2C slave device on the same bus has a unique 7-bit address. In our example, the addresses of the two PD9593 chips are 0x20 and 0x21. Table 5.4 shows that the chip has eight internal registers. As mentioned in Section 2, after the master device successfully sends the address byte, it will send another byte to select the internal register. Among these internal registers, the configuration register is responsible for controlling the directions of the IO pins. Zero means the corresponding pin is output. Because the PLC uses PD9593 to control the LED, all the pins are outputs. The input port registers reflect the pins' incoming logic levels, regardless of whether the pin is defined as an input or an output by the configuration register. Nevertheless, we do not need to read the input registers.

Figure 5.10 shows the pseudo-code to initialize the I2C IO expander using the ROM

³https://github.com/whensungoesdown/at45db021_teensy32

CMD Byte	Register	Protocol	Power-up Default
0x00	Input Port 0	Read byte	xxxx xxxx
0x01	Input Port 1	Read byte	xxxx xxxx
0x02	Output Port 0	Read/Write byte	1111 1111
0x03	Output Port 1	Read/Write byte	1111 1111
0x04	Polarity Inversion Port 0	Read/Write byte	0000 0000
0x05	Polarity Inversion Port 1	Read/Write byte	0000 0000
0x06	Configuration Port 0	Read/Write byte	1111 1111
0x07	Configuration Port 1	Read/Write byte	1111 1111

Table 5.4: The internal registers of the PD9593 are selected by the master device sending the command byte. To control the LED, we need to operate the control register and the output register and keep the two polarity inversion registers' default value.

API. It first enables the I2C master device and sets the clock frequency to be the same as the microcontroller. After that, call `ROM_I2CMasterSlaveAddrSet()` to select the target device's address, whether to send or receive data. Then call `ROM_I2CMasterDataPut()` to set the data to be sent next. Only after calling `ROM_I2CMasterControl()`, the data will be sent out. In our example, two bytes with content zero are sent to Configuration Port0 and Port1. This operation sets a total of 16 pins as output mode.

```

ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
ROM_GPIOPinTypeI2C(0x40059000, 0xC);
clock = ROM_sysCtlClockGet();
ROM_I2CMasterInitExpClk(0x40020000, clock, TRUE);
ROM_I2CMasterSlaveAddrSet(I2C0, 0x21, FALSE);
ROM_I2CMasterDataPut(I2C0, 0x6);
ROM_I2CMasterControl(I2C0, I2C_MASTER_CMD_BURST_SEND_START);
while (ROM_I2CMasterBusy(I2C0)) {
};
ROM_I2CMasterDataPut(I2C0, 0x0);
ROM_I2CMasterControl(I2C0, I2C_MASTER_CMD_BURST_SEND_COUNT);
while (ROM_I2CMasterBusy(I2C0)) {
};
ROM_I2CMasterDataPut(I2C0, 0x0);
ROM_I2CMasterControl(I2C0, I2C_MASTER_CMD_BURST_SEND_FINISH);
while (ROM_I2CMasterBusy(I2C0)) {
};

```

Figure 5.10: The pseudo-code to initialize the I2C IO expander is reversed-engineered from the firmware. To make it more intuitive, we use some macro definitions as parameters. These macros' actual value is not difficult to find in the header files released by the vendor. The code is provided to help understand how to control the LED lights on the PLC through the I2C bus.

After initialization, Figure 5.11 shows how to control the LEDs. For example, we

send two bytes with content 0xFF to the Output Port0 and Port1, lighting up 16 LED lights controlled by this device.

```
ROM_I2CMasterSlaveAddrSet(I2C0, 0x21, FALSE);
ROM_I2CMasterDataPut(I2C0, 0x2);
ROM_I2CMasterControl(I2C0, I2C_MASTER_CMD_BURST_SEND_START);
while (ROM_I2CMasterBusy(I2C0)) {
};
ROM_I2CMasterDataPut(I2C0, 0xFF);
ROM_I2CMasterControl(I2C0, I2C_MASTER_CMD_BURST_SEND_CONT);
while (ROM_I2CMasterBusy(I2C0)) {
};
ROM_I2CMasterDataPut(I2C0, 0xFF);
ROM_I2CMasterControl(I2C0, I2C_MASTER_CMD_BURST_SEND_FINISH);
while (ROM_I2CMasterBusy(I2C0)) {
};
```

Figure 5.11: Each pin in the Output register controls an LED light separately, so two 0xFF bytes can control 16 LEDs. If we only want to control a few of the LED lights in one register, we can call `ROM_I2CMasterControl()` with parameter `I2C_MASTER_CMD_SINGLE_SEND`.

Onboard Connectors. There are several connectors on the real-time module **C**, as shown in Figure 5.12. P607 connects to the communication module **B** and P609 connects to the power module **A**. Usually, we think that the module **A** is just a power module responsible for providing three volts to others. However, this is a good place for purposefully damage the PLC by a short circuit, where the power flow is large.

By conducting the connectivity tests with a multimeter, we perceive some pins' function in P607 as shown in Figure 5.13. As mentioned earlier, the I2C bus passes through this connector. Besides, the CAN bus also passes through this connector. In this way, both the communication module and the real-time module are connected to the CAN bus connector in the bottom right corner in Figure 5.12. These two modules can not only control the external CAN bus device, but they can also communicate via the CAN bus themselves. However, the communication module has a higher priority. Setting pin29 on the P607 connector can block the transmission of the signal on pin21, the CAN bus's Rx/D signal line. On the microcontroller side, the master device CAN0 uses the PA6 and PA7 pins.

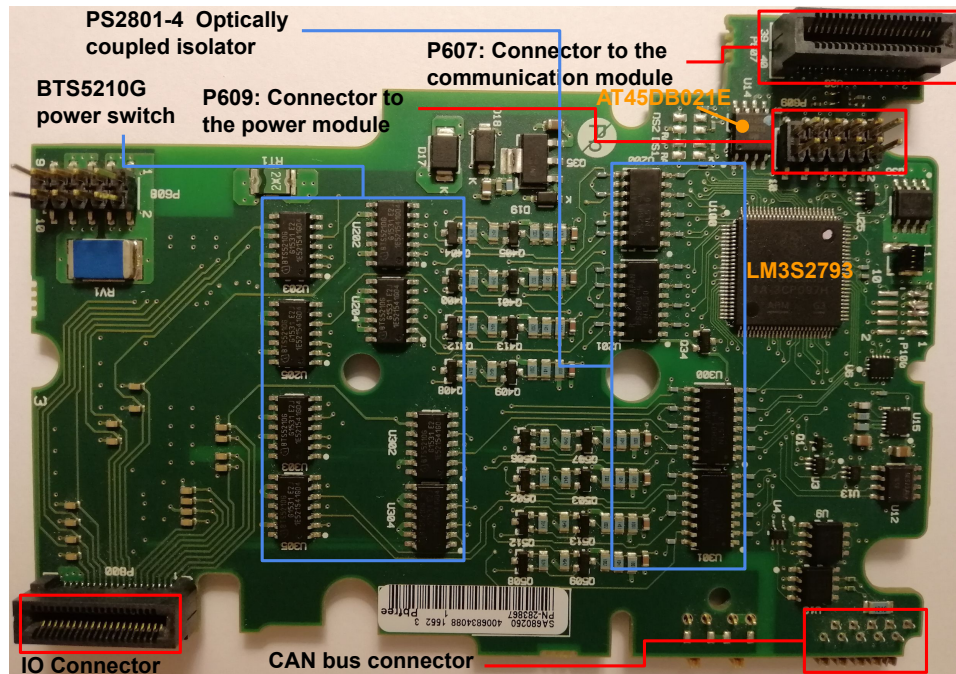


Figure 5.12: The real-time module reads the input signals, runs the ladder logic, and then drives the output signal according to the result. Therefore this module needs to be connected to all other modules. In addition to the microcontroller and the flash memory, there are power regulators and IO chips on this module. The optically coupled isolators prevent high voltages from affecting the microcontroller when receiving the signal, and the power switches provide the electrical connection between the output pin and the voltage source.

4.2 Design

Typically, the PLC has a dedicated real-time microcontroller that handles IO, in this case, module **D**. It runs minimal code, mainly the compiled ladder logic. To receive ladder logic update, the communication module **C** talks to the HMI and then update the real-time microcontroller through the CAN bus. Therefore, to remotely control the PLC's IO, the attacker needs to control further the communication module **C**, which itself is an independent system usually with a more powerful processor. Moreover, an abnormal traffic filtering mechanism [82] also may be deployed to protect ICS networks. Even though we successfully controllers all the embedded system in the PLC and penetrates the network traffic monitors, we still have to face the challenge that a critical infrastructure may run on an isolated local network. Therefore, we choose to use a separate network using GSM.

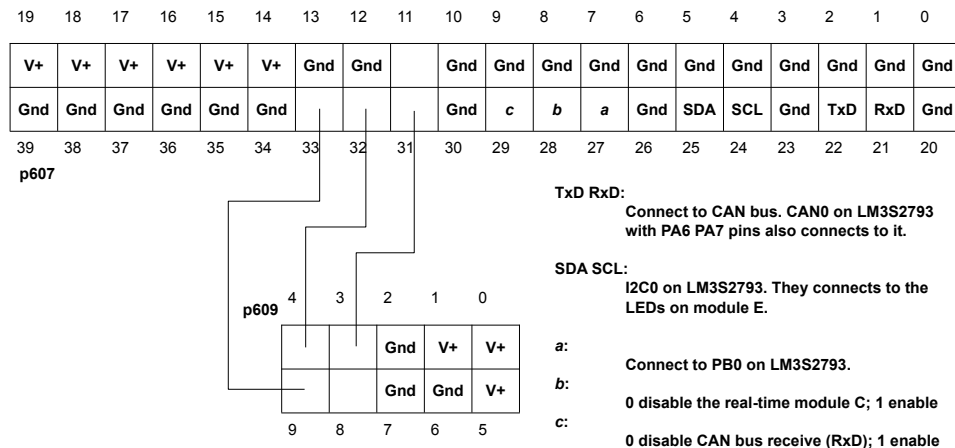


Figure 5.13: The connector between the communication board (module **B**) and real-time board (module **C**). There are many pins, but the majority of them are power and ground. The microcontroller controls the LED module through the P607 connector. The communication module can block the operation of the entire real-time module through pin28. However, there are still a few pins that we have not figured out their functions. We consider it as one of our further works.

The SIM800C is a Quad-Band GSM/GPRS module. It has strong extension capability with interfaces including UART, USB2.0, and GPIO. Figure 5.14 shows that the SIM800C module connects with the Teensy board through a serial port. First, the Teensy board initializes the cellular module using AT command, connecting to the network. Once a text message is received, the Teensy board reads it and looks for control commands. In such a case, the command will be parsed as IO operations that eventually turn into particular memory read/write on PLC's GPIO port.

5 Implementation and Experiment

This section provides the implementation details and discusses the issues that we encountered during the development.

Read/Write Memory. We briefly introduced the JTAG protocol in the background (Section 2) earlier. Our driver sends instructions to the IR and reads the returned result according to the JTAG state machine. Nevertheless, we need to know how to operate ARM's CoreSight components for this particular hardware backdoor prototype. We choose the JTAG interface instead of SW (Serial Wire), so the corresponding debug port is JTAG-DP or SWJ-DP, as shown in Figure 5.2.

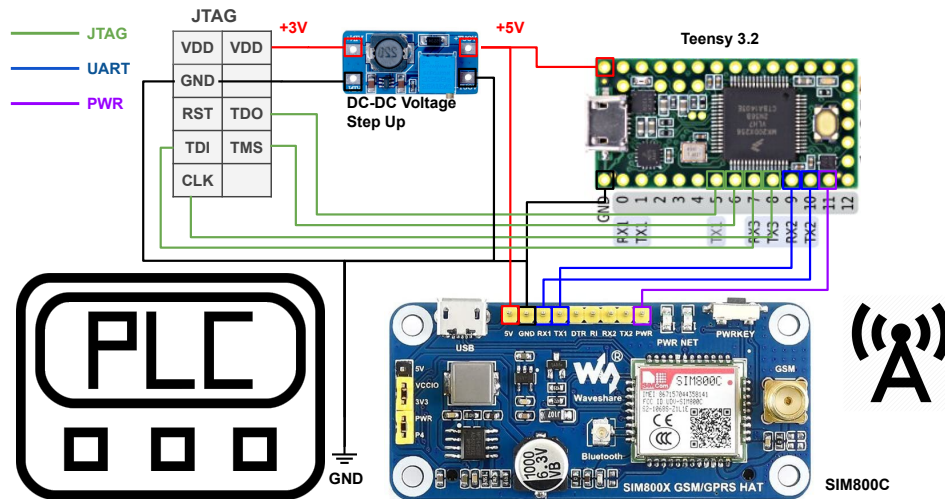


Figure 5.14: The power can be drawn from the JTAG pad or directly from the border connector p607. Since the JTAG pad provides the 3 volts source, we also need a DC-DC voltage step-up module that boosts from 3 volts to 5 volts. One GPIO pin from Teensy connects the PWR pin of the SIM800C board. It is used to deliver the power and reset sequence. After SIM800C initialized, the Teensy module sends AT commands through the serial port.

IR value	JTAG-DP Register	DR width	Description
b1000	ABORT	35	JTAG-DP Abort Register
b1010	DPACC	35	JTAG DP Access Register
b1011	APACC	35	JTAG AP Access Register
b1110	IDCODE	32	JTAG Device ID Code Register
b1111	BYPASS	1	JTAG Bypass Register

Table 5.5: DPACC is used for Debug Port (DP) accesses. APACC is used for Access Port (AP) accesses, and it can access a register of a debug component of the system to which the interface is connected.

Through DPACC and APACC registers, the debugger can access resources provided by other access ports (AP). As mentioned earlier, an access port provides the interface between the debug port interface and one or more debug components present within the system. There are two kinds of access ports: Memory Access Ports (MEM-AP) and JTAG Access Ports (JTAG-AP), and MEM-AP is designed for connects to memory bus system with address and data controls. Since our backdoor wants to access memory and GPIO, we need to access either AHB-AP or the MEM-AP, which handled the differences between the underlying bus.

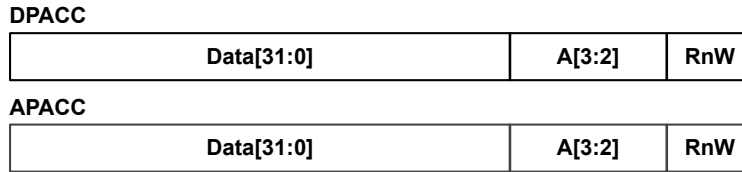


Figure 5.15: Both of the two registers are 35 bits long and can be scanned in/out through the JTAG protocol with IR instruction b1010 and b1010, respectively. RnW takes one bit, and zero indicates a write. A[3:2] selects the register within a bank.

Offset	Register	Description
0x00		Reserved
0x04	CTRL/STAT	Control and State Register
0x08	SELECT	AP Select
0x0C	RDBUFF	Read Buffer

Table 5.6: Debug Port registers. Debug Port only has one bank, a total of four registers, which can be specified by A[3:2] of the DPACC register.

Figure 5.15 shows that both DPACC and APACC have the same structure, and they can be scan in/out through the JTAG state machine with the specific IR instruction value. The lowest bit indicates whether to read or write DP/AP, where zero means write.

According to ARM Debug Interface Architecture Specification v5.0 to v5.2, every bank has four registers, and A[3:2], the two bits are used to select the register from the bank. The DP only has one bank, and the MEM-AP has 16 banks, as listed in Table 5.6 and Table 5.7, respectively. To select the AP's bank, we also need to write the bank address into the DP's SELECT register.

Offset	Register	Description
0x00	CSW	Control/Status Word register
0x04	TAR	Transfer Address Register
0x08		Reserved
0x0C	DRW	Data Read/Write register
...		
0xFC	IDR	Data Identification register

Table 5.7: Part of Memory Access Port (MEM-AP) registers. MEM-AP has 16 banks, and each bank has four registers, which can be specified by A[3:2] of the APACC register. The bank needs to be specified by the DP:SELECT register.

To read and write memory, we need to use the internal registers provided by the MEM-AP. Specifically, we need to sue CSW, TAR, DRW, and others. Since these registers are in the MEM-AP's bank 0, we must first use DPACC to select it. Take writing memory as an example. Next, we need to write the destination address to TAR and then write the value to DRW. Figure 5.16 shows a pseudo-code for writing memory.

```

TAP_Idle();
// shift to DPACC
TAP_ShiftIR(DPACC);
// select DP.CTRL/STAT, scan in data
TAP_ShiftDR(DP.CTRL/STAT, CSYSPWRUPREQ | CDBGPWRUPREQ);
// scan out status
TAP_ShiftDR(status);
// select DP.SELECT, set AP bank0
TAP_ShiftDR(DP.SELECT, AP_BANK0);

// shift to APACC
TAP_ShiftIR(APACC);
// select AP.CSW, set write size
TAP_ShiftDR(AP.CSW, SIZE_WORD);
// select AP.TAR, set destination address
TAP_ShiftDR(AP.TAR, address);
// select AP.DRW, write data
TAP_ShiftDR(AP.DRW, value);

```

Figure 5.16: Pseudo-code for memory writing through MEM-AP. Notice, CSW, TAR and DRW are all in the bank zero. However, the bank is specified in the DP.SELECT instead of a register in AP.

Send/Receive SMS Message. SIM800C provides a serial port as the interface to receive AT commands. When HARDDOOR is started, we use the AT+CMGF command to set the GSM chip in SMS Text Mode. Then we use the AT+CNMI command to set how to notify when new messages come. After that, the Teensy board keeps checking the serial port for new messages every second. If there is one, read the content and parse if it is a pre-defined attack command.

To send out a message, use the AT+CMGS command to set the destination phone number and then send the text message to the serial port.

6 Evaluation

In this section, we evaluated several aspects of HARDDOOR. Since it is a hardware backdoor, we need to measure its physical dimensions and appearance. We also evaluated the effects of HARDDOOR on the PLC, including the impacts on the performance,

memory storage, and power consumption.

The model PLC is the Allen Bradley 1769-L18ER-BBIB CompactLogix 5370, which is equipped with a TI Stellaris LM3S2793 microcontroller. The microcontroller is based on ARM Cortex-M3 architecture and operates at 80MHz. It has 128KB single-cycle flash memory and 64KB single-cycle SRAM. The PLC has 16 DC digital inputs and 16 DC digital outputs, which eventually corresponds to the microcontroller's GPIO port.

Dimensions and appearance. We use commercial off-the-shelf modules to build the prototype, namely the Teensy 3.2 development board and Waveshare SIM800C HAT. They are not the smallest in size. For example, the SIM800C in SiP packaging with a minimal PCB board is much smaller. However, the SiP needs a voltage of 3.7 volts which a Lithium-Ion battery usually provides. GSM is an impulse type transmitting power, even though the average current is low, but the instantaneous current can reach more than 1.5A, so the board's external power supply is necessary.

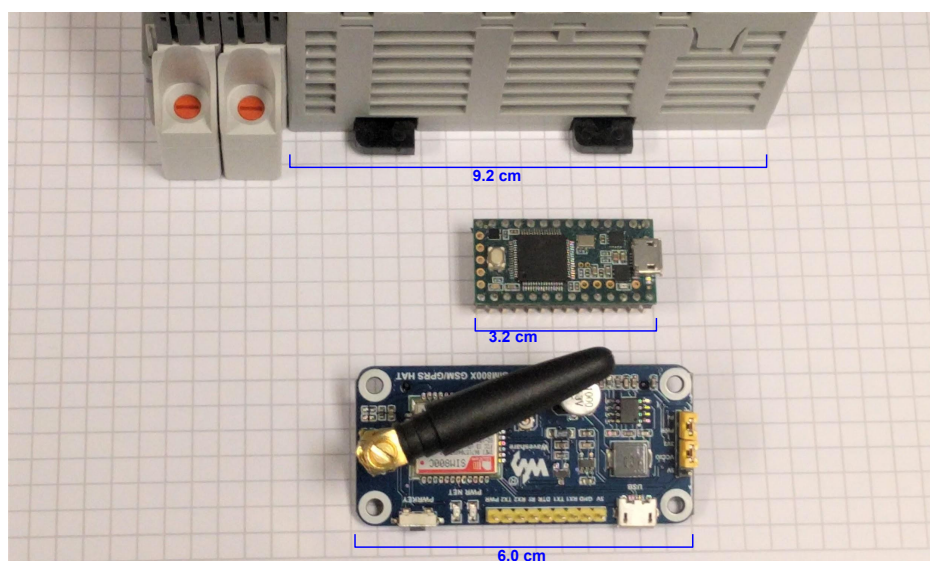


Figure 5.17: The Allen Bradley 1769-L18ER-BBIB CompactLogix 5370 PLC is in a 9.2cm x 13cm rectangle shape with sufficient space to contain the two boards and extra wires and connectors. The SIM800C HAT takes a full-size sim card, and the antenna takes much space.

Figure 5.17 shows the physical size of the two boards compared to the Allen Bradley PLC. Notice that the two main chips on each board are relatively small. Especially some WIFI chips [12] [10] also provide a microcontroller for general-purpose tasks,

which the two chips can combine. In that case, the hardware can further reduce its size. However, we can not shrink its physical size as small as invisible. We argue that the more effective way to hide the hardware implant is to make it easily overlooked. It means to use customized PCB with the same color and same style connector attaching to the PLC's board. Because until today, there is very little information publicly available regarding those widely deployed PLCs. Some good examples are the separate IPMI (Intelligent Platform Management Interface) [149] and KVM (Keyboard, Video, and Mouse) [80] modules that usually mounted on the server's motherboards.

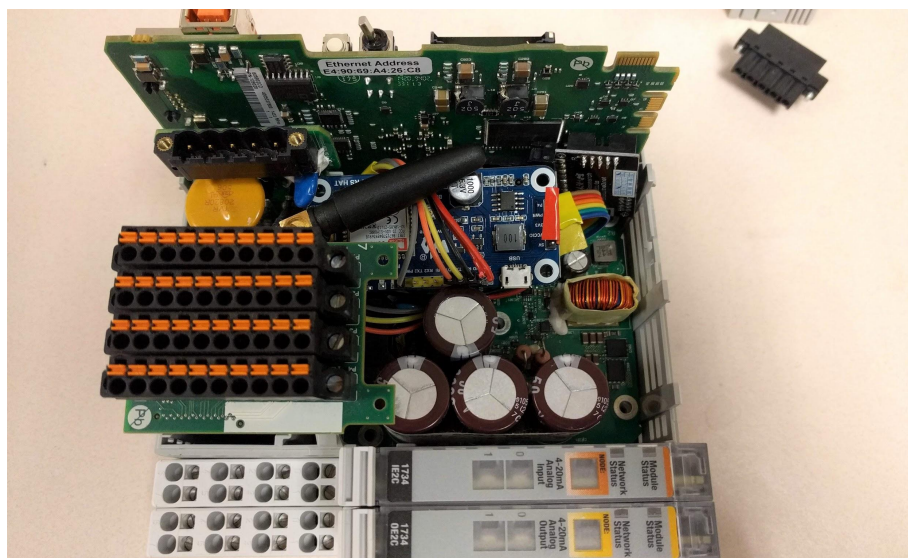


Figure 5.18: We cover the necessary parts using tape to prevent a short circuit. The HARDDOOR connects to the PLC's microcontroller board's JTAG pad through a 10-pin socket and a long cable. The antenna for receiving GSM signal is also stuffed inside the PLC.

Figure 5.18 shows that after wiring two boards and wrapping them with tape to prevent a short circuit, it is small enough to fit inside the PLC's plastic case. We connect the JTAG pads with GPIO pins from the Teensy board. One of the PLC we use for this experiment turns out to have a 10-pin socket soldered on the pad, but all other PLCs we own do not have such a setting. A novel design for easy installation avoiding soldering in the field and reliable connection with the JTAG pad is necessary. We consider this as one of our further works.

As shown in Figure 5.20, the hardware implant is not noticeable from the PLC's

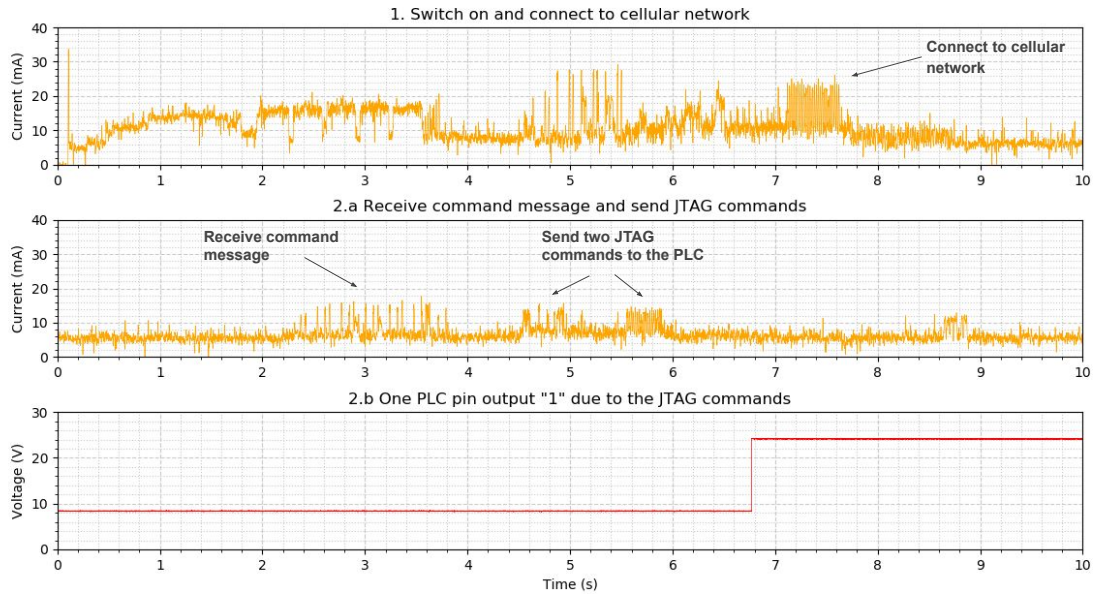


Figure 5.19: The hardware implant does not consume a lot of power, and the power consumption will only increase slightly when starting and executing the attack command. Sub-figure 1 shows the power consumption during the startup. Sub-figure 2.a shows the power consumption during an demo attack. 2.b indicates an output pin of the attacked PLC.

outside appearance; no parts are exposed, it only seems to take up some space. However, heat dissipation may be a problem to be considered in the future.

Performance. Since this backdoor is implemented on the hardware level, it almost cost zero performance overhead. For instance, if we choose to change IO through override signals transmitting in the low-speed bus, pull-up or pull-down the voltage level, there will be no overhead added to the microcontroller. On the other side, if we use the JTAG interface instead, it may cause a slight performance overhead based on the microarchitecture. Due to various implementations, JTAG debugging capabilities can be intrusive or non-intrusive. The conventional JTAG debug is invasive, which halt the processor using breakpoints and watchpoints. It also needs to halt the processor before it can modify any register.

Nevertheless, the debug functionality provided in LM3S2793 is as CoreSight components. It provides real-time access for the debugger without halting the processor to AMBA system memory, peripheral registers, and all debug configuration registers.

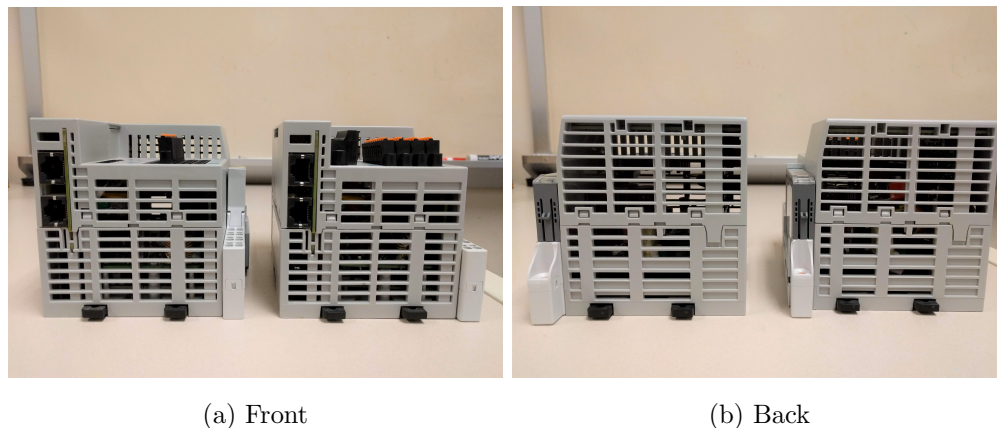


Figure 5.20: The two figures are the Allen Bradley 1769-L18ER-BBIB CompactLogix 5370 PLC's front and back view, in which both the PLC on the right has the hardware implant installed. There is no significant difference from the appearance point of view, and no parts are exposed to the outside.

Therefore HARDDOOR can modify the GPIO through the AHB bus without any software overhead.

Also, the external or timer interrupt may be delayed for a few clock cycles when encountering JTAG related operations. However, HARDDOOR has no operation when in standby mode, and controls on IO only take a few memory reads/writes. Furthermore, we regulate our attacking operations at a low pace not to jam the system. The resulting performance overhead is generally negligible, as shown in Figure 5.21.

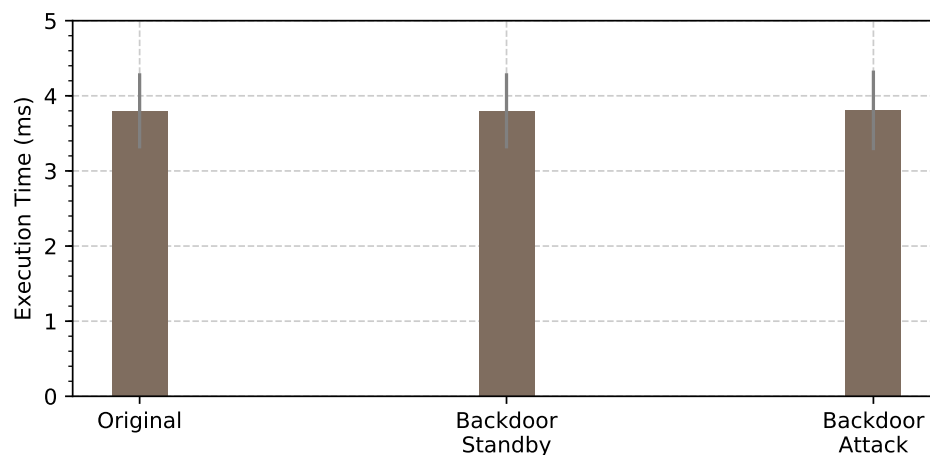


Figure 5.21: To accumulate execution time, we use a counter in the ladder logic and use an output signal to indicate the begin/end time of the test. During the standby test, HARDDOOR has attached the PLC, but no command is sent. We alter the PLC's output every 500ms to imitate a malicious operation for the attack test.

Memory Consumption. HARDDOOR neither occupies flash memory to reside on the system nor take SRAM during the runtime. Even when it is controlling IO, the operations are performed on memory-mapped IO for GPIO ports. Therefore, the memory consumption for HARDDOOR is zero.

Power Consumption. Because of the extra circuitry we bring to the system, the power consumption of the PLC increases. Although the two embedded devices consume very little power compared to the field power that the PLC provides, this is an anomaly we brought into the system. Therefore we measure it with the PLC that is connected to the field power but carries no applicants. We connect a resistor in series to the power supply line of the PLC and use an oscilloscope to measure the voltage across the resistor. Figure 5.19 shows that several slight current peaks occur when the cellular chip is searching and connecting to the GSM network and when HARDDOOR receives the SMS message and sending JTAG commands to the PLC.

SMS Message. One concern we have is that since the antenna is also stuff inside the PLC's case, the signal strength may not be good. So we test it for receiving various lengths of SMS messages, as shown in Figure 5.22. We use a phone to send the command messages to HARDDOOR, and we send each command 20 times to calculate the average. The cellular networks we used are T-Mobile and Google Fi. Although the time of message transmission is not very accurate, it is reliable. Different lengths of information have different purposes. For starting a denial-of-service attack or a pre-defined function, one byte is enough. To change specific IO, we need a few bytes to descript its address or index. We also test receiving hundreds of bytes. It can be used to update the firmware on the PLC's flash remotely.

7 Related Work

Firmware modification attacks [123] [14] [21] [39] [84] [140] constitute significant attacks targeting embedded systems, industrial control systems, and IoT devices.

Harvey [55] is a physical-aware stealthy rootkit against a cyber-physical power grid control system. It hides within the PLC's firmware below the control logic and modifies

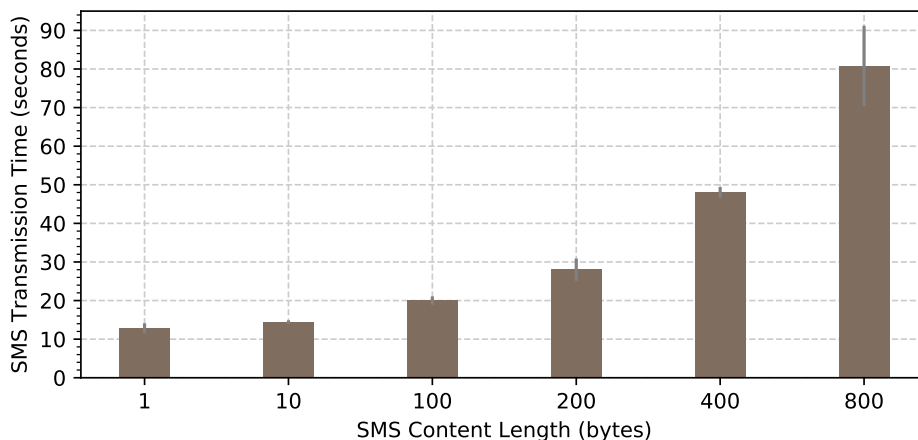


Figure 5.22: The GSM network may be congested when there are many users around. The large piece of message, such as 800 bytes, will be segmented into several packets, and the transmission time varies, but still, the received order and the message content are correct.

control commands before sending it to the physical plant's actuators. This work [112] implements a malicious firmware that ignored incoming print commands for a printed 3D model, substitutes malicious print commands for an alternate 3D model. If the firmware attacks can be carried out remotely, the harm will be even more significant. Cui et al. [39] gives a detailed case study of the HP-RFU (Remote Firmware Update) LaserJet printer firmware modification vulnerability, which allows arbitrary injection of malware into the printer's firmware via standard printed documents.

These systems have one thing in common: they all run on a microcontroller with limited computing power. Therefore, these devices run a simple real-time operating system, and there has not been a complete anti-virus system or a series of integrity verification features and programs provided by hardware and operating system like those on modern PCs. Due to the lack of security features, the programs run on those systems are often more vulnerable. Moreover, because they usually focus on specific areas and are not easily accessible, security issues are ignored. However, once those systems are compromised, through firmware modification, the attacker can stay in the dark for a long time without being discovered and cause a significant impact at a particular moment.

On the other hand, there are many ways to protect the firmware from being modified.

ConFirm [163] is a low-cost technique to detect malicious modifications in the firmware of embedded control systems. It measures the number of low-level hardware events that occur during the execution of the firmware. Lee et al. [93] presented a technique for binding software to hardware instances that use the devices' hardware security properties. The proposed technique assures manufacturers that only they can perform their hardware and software binding and create their products.

Remote software attestation [96] is also a defense against firmware modification. SWATT [145] verifies embedded devices' memory contents and establishes the absence of malicious changes to the memory contents without using extra security hardware features. It uses a challenge-response protocol between the verifier and the embedded device. The verifier sends a challenge to the embedded device. The embedded device computes a response to this challenge in a pre-defined protocol between the verifier and the device. The device can only give the correct answer if the memory content is intact. Otherwise, the attacker has to know the verifier's secret algorithm to break the verification. Similarly, SBAP [95] also provides a software-only solution to verify the firmware integrity but with the help of an existing peripheral device.

Other methods such as firmware binary obfuscation [42] [139] [31], makes it very challenging for firmware modification attacks. It requires comprehensively analyzing each device to find a suitable place to inject malicious code.

8 Discussions and Mitigations

Reducing the size of the hardware implant is necessary but not the main factor in disguising. A customized PCB board allows all the required chips to be installed together so that there is no Dupont jumper wire, which makes the hardware implant very suspicious. The SIM800C chip already is a SiP (System in a Package), and the JTAG driver can run on a minimal core such as cortex-M0. Combining these two can make the backdoor simpler, and we can already find such WIFI chips, such as ESP32 [129], as of the time of writing this paper. Moreover, a better camouflage makes the hardware backdoor look like a legit accessory of the PLC. The IPMI (Intelligent Platform

Management Interface) module is a good example. It needs to be purchased separately and installed on the reserved socket on the server motherboard to activate the remote control function.

To mitigate a hardware backdoor attack such as HARDDOOR, we first measure the power usage to find the overhead caused by the extra circuitry, as evaluated in Section 6. The result shows a few current peaks when HARDDOOR connects to the GSM network and sends JTAG commands to PLC. However, its power consumed is not much, a few milliamperes. Moreover, the PLC’s power consumption is also constantly fluctuating, and setting a threshold with little redundancy will affect the system’s reliability.

In particular, to avoid malicious use of the JTAG interface, the manufacturer can blow the corresponding physical JTAG fuse at the factory [137] [25]. Blowing the fuse completely disables the JTAG port and is not reversible.

However, by tapping the open wire on the boards, we can directly control each peripheral device because they are connected to the microcontroller through various buses. We think that some physical protection will cause trouble to the attacker. For example, the Chip-on-Board (COB) [89] packaging with a black blob for low-cost IC items makes it more challenging to identify the chip underneath. Nevertheless, we believe it is not a good practice to cover the whole board, and heat dissipation is critical for large area ICs. One possible direction for preventing bus signal hijacking is to send packet-based encrypted data, which requires the microcontroller and peripherals to exchange encryption keys and maintain a connection state. However, it may not be practical for low-speed devices and low-end microcontrollers.

9 Conclusions

To achieve more stealthiness in a sophisticated APT attack, we think the trend is that the trojan moves towards the hardware level, especially with the emerging supply chain attacks. We present HARDDOOR, a parasitical hardware implant that directly controls the PLC through wire and bus signal hijacking. HARDDOOR does not modify the firmware nor relies on PLC’s network communication. It can manage/damage the ICS

system's physical assets by controlling PLC's IO. At the same time, it provides a faked expected view of the system to circumvent detections. Our prototype is small in size, and the experiment results demonstrate the feasibility of HARDDOOR in practice.

Chapter 6

Conclusion

This dissertation presents novel system memory protection techniques to address several state-of-the-art vulnerability challenges and bring a new class of attack.

First, the author aims at mitigating heap overflow vulnerabilities. In a heap overflow attack, the attacker often abuses the heap metadata to exploit the system. Therefore, the author presents a novel heap allocator that protects heap metadata with a secure environment in the user space, using the hardware feature Intel Software Guard Extensions (SGX). SGX provides separated memory spaces, so-called enclaves, to protect critical user data. With the heap metadata reorganized and protected, the mitigation prevents vulnerabilities from abusing metadata, hence stop attacks of such kind.

Next, the author aims at mitigating use-after-free (UAF) vulnerabilities. UAF is especially prevalent in the world of web browsers. Despite many successful UAF exploits against widely-used applications, state-of-the-art defense mechanisms have proved to leave the systems vulnerable still. The author argues that a successful UAF exploit is feasible because of the fine-grained determinism provided by existing heap memory allocators. Therefore, the author presents a new defense strategy that leverages additional memory buffers to make allocation outcomes locally unpredictable to adversaries. It significantly lowers the success rate of a UAF exploit.

Then, the author aims at mitigating kernel-level time-of-check-to-time-of-use (TOC-TOU) vulnerabilities. Kernel-level TOCTOU widely exists in operating systems, especially Microsoft Windows. When serving a system call, the kernel inevitably gets parameters from the userspace. Read the same user-mode variable repeatedly may lead to data inconsistency under a race condition between the kernel and userspace. The

core of this mitigation is to use Supervisor Mode Access Prevention (SMAP), a hardware feature, to detect kernel access to userspace data. Due to the Windows system's complex nature, the author further develops a lightweight hypervisor to contain the system-wide hardware feature SMAP into specific processes to prevent deadlock caused by nested SMAP exceptions. It is the first runtime mitigation technique on Windows with acceptable performance overhead (less than 10

Last, the author tries to think out of the box, present a new class of attack that deploys a small-size parasitical hardware implant attached to a victim programmable logic controller (PLC). It controls the PLC by modifying the digital signal or hijacking the various buses on the boards. This attack can be deployed either during the supply chain or stealthily installed in remote plants. The author makes such a point, although software-level weakness mitigation and protection make attacks more difficult to succeed, lower-level and new types of attacks will be more challenging to detect and protect in the software domain, and this is a real threat.

Bibliography

- [1] M. Abadi, A. Birrell, T. Harris, and M. Isard, “Semantics of Transactional Memory and Automatic Mutual Exclusion,” *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 63–74, 2008.
- [2] H. Abdul-Aziz, Z. Simon, and G. Brian, “Abusing silent mitigations,” *Blackhat US*, 2015.
- [3] T. Alves and D. Felton, “Trustzone: Integrated Hardware and Software Security,” *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.
- [4] O. Analytica, “Solarwinds hack will alter us cyber strategy,” *Emerald Expert Briefings*, no. oxan-db, 2021.
- [5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, “Innovative technology for cpu based attestation and sealing,” in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.
- [6] S. Andersen and V. Abella, “Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies,” 2004.
- [7] Anonymous, “Once upon a free,” *Phrack Magazine*, vol. 57, no. 9, 2001.
- [8] Argp and Huku, “Pseudomonarchia jemallocum,” *Phrack Magazine*, vol. 68, no. 10, 2012.
- [9] P. Argyroudis and C. Karamitas, “Exploiting the Jemalloc Memory Allocator: Owing Firefox’s Heap,” *Blackhat USA*, 2012.
- [10] N. Arstenstein, “Broadpwn: Remotely compromising Android and iOS via a bug in Broadcom’s Wi-Fi chipsets,” *Black Hat USA*, 2017.
- [11] E. Ashfield, “Serial wire debug and the coresighttm debug and trace architecture eddie ashfield, ian field, peter harrod, sean houlihane, william orme and sheldon woodhouse arm ltd 110 fulbourn road, cambridge, cb1 9nj, uk,” 2006.
- [12] M. Babiuch, P. Foltýnek, and P. Smutný, “Using the ESP32 microcontroller for data processing,” in *2019 20th International Carpathian Control Conference (ICCC)*. IEEE, 2019, pp. 1–6.
- [13] R. Barry *et al.*, “Freertos,” *Internet*, Oct, 2008.
- [14] Z. Basnight, J. Butts, J. Lopez Jr, and T. Dube, “Firmware modification attacks on programmable logic controllers,” *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76–84, 2013.

- [15] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
- [16] —, “Tiny Code Generator,” 2009.
- [17] E. D. Berger and B. G. Zorn, “Diehard: probabilistic memory safety for unsafe languages,” in *Acm sigplan notices*, vol. 41, no. 6. ACM, 2006, pp. 158–168.
- [18] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address obfuscation: An efficient approach to combat a broad range of memory error exploits.” in *Usenix Security*, vol. 3, 2003, pp. 105–120.
- [19] M. Bishop, “Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux,” *Technical Report CSE-95-98*, 1995.
- [20] M. Bishop, M. Dilger *et al.*, “Checking for Race Conditions in File Accesses,” *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996.
- [21] A. Blanco and M. Eissler, “One firmware to monitor’em all,” 2012.
- [22] M. D. Bond, K. E. Coons, and K. S. McKinley, “PACER: Proportional Detection of Data Races,” in *ACM Sigplan Notices*. ACM, 2010.
- [23] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, “Fixing races for fun and profit: How to abuse atime,” in *USENIX Security Symposium*, 2005.
- [24] M. Bozdal, M. Samie, and I. Jennions, “A survey on CAN bus protocol: attacks, challenges, and potential solutions,” in *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE, 2018, pp. 201–205.
- [25] R. F. Buskey and B. B. Frosik, “Protected JTAG,” in *2006 International Conference on Parallel Processing Workshops (ICPPW’06)*. IEEE, 2006, pp. 8–pp.
- [26] D. U. Case, “Analysis of the cyber attack on the ukrainian power grid,” vol. 388, 2016.
- [27] C. Cesar, “Easy local Windows Kernel exploitation,” *Black Hat US*, 2012.
- [28] L. Chen, D. Yue, C. Dou, J. Chen, and Z. Cheng, “Study on attack paths of cyber attack in cyber-physical power systems,” *IET Generation, Transmission & Distribution*, vol. 14, no. 12, pp. 2352–2360, 2020.
- [29] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-Control Data Attacks Are Realistic Threats,” in *USENIX Security Symposium*, vol. 5, 2005.
- [30] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, “Savior: Towards Bug-Driven Hybrid Testing,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1580–1596.
- [31] X. Cheng, Y. Lin, D. Gao, and C. Jia, “DynOpVm: VM-based software obfuscation with dynamic opcode mapping,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2019, pp. 155–174.

- [32] A. Cherepanov and R. Lipovsky, “Blackenergy—what we really know about the notorious cyber attacks,” *Virus Bulletin October*, 2016.
- [33] B.-C. Choi, S.-H. Lee, J.-C. Na, and J.-H. Lee, “Secure firmware validation and update for consumer devices in home networking,” vol. 62, no. 1. IEEE, 2016, pp. 39–44.
- [34] R. Chris, “Partitionalloc - a shallow dive and some rand; available at <https://struct.github.io>,” 2016.
- [35] C. Click, “Azul’s Experiences with Hardware Transactional Memory,” in *HP Labs-Bay Area Workshop on Transactional Memory*, vol. 89, 2009.
- [36] M. Conover, “w00w00 on heap overflows.” 1999.
- [37] J. Corbet, “Supervisor Mode Access Prevention; available at <https://prod.lwn.net/Articles/517475>,” 2012.
- [38] T. D. Cottle and T. T. Spits, “Programmable interrupt controller with interrupt set/reset register and dynamically alterable interrupt mask for a single interrupt processor,” 2001, uS Patent 6,263,396.
- [39] A. Cui, M. Costello, and S. Stolfo, “When firmware modifications attack: A case study of embedded exploitation,” 2013.
- [40] CVE, “CVE-2013-1254; available at <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-1254>,” 2013. [Online]. Available: <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-1254>
- [41] Cvedetails, “Common vulnerability exposure (CVE) details; available at <http://www.cvedetails.com>,” 2017.
- [42] B. Cyr, J. Mahmood, and U. Guin, “Low-cost and secure firmware obfuscation method for protecting electronic systems from cloning,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3700–3711, 2019.
- [43] D. Dai Zovi, “Practical return-oriented programming,” *SOURCE Boston*, 2010.
- [44] D. Dean and A. J. Hu, “Fixing Races for Fun and Profit: How to Use access (2),” in *USENIX Security Symposium*, 2004, pp. 195–206.
- [45] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, “Early Experience with a Commercial Hardware Transactional Memory Implementation,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 157–168.
- [46] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, “The matter of heartbleed,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC ’14. New York, NY, USA: ACM, 2014, pp. 475–488.
- [47] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, “Dynamic Binary Translation and Optimization,” *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 529–548, 2001.

- [48] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, “Effective Data-Race Detection for the Kernel,” in *OSDI*, 2010.
- [49] S. Etigowni, D. Tian, G. Hernandez, S. Zonouz, and K. Butler, “CPAC: securing critical infrastructure with cyber-physical access control,” in *Proceedings of the 32nd annual conference on computer security applications*, 2016, pp. 139–152.
- [50] exploit-db.com, “Structured Exception Handler Exploitation,” 2002.
- [51] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet, “Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free,” in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 2016, pp. 1–12.
- [52] S. Fischer, “Supervisor Mode Execution Protection,” in *NSA Trusted Computing Conference and Exposition*, 2011.
- [53] C. Flanagan and S. N. Freund, “FastTrack: Efficient and Precise Dynamic Race Detection,” in *ACM Sigplan Notices*. ACM, 2009.
- [54] D. Flynn, “AMBA: enabling reusable on-chip designs,” *IEEE micro*, vol. 17, no. 4, pp. 20–27, 1997.
- [55] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz, “Hey, my malware knows physics! attacking plcs with physical model aware rootkit.” in *NDSS*, 2017.
- [56] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler, “Using Hardware Transactional Memory for Data Race Detection,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–11.
- [57] H. Haes Alhelou, M. E. Hamedani-Golshan, T. C. Njenda, and P. Siano, “A survey on power system blackout and cascading events: Research motivations and challenges,” *Energies*, vol. 12, no. 4, p. 682, 2019.
- [58] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski *et al.*, “The IBM Blue Gene/Q Compute Chip,” *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.
- [59] B. Hawkes, “Attacking the vista heap,” *Blackhat USA.(Aug. 2008)*, 2008.
- [60] —, “Attacking the Vista Heap,” *Blackhat USA*, 2008.
- [61] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-free Data Structures,” in *Proceedings of the 20th annual international symposium on computer architecture*, 1993, pp. 289–300.
- [62] H. Högl and D. Rath, “Open On-Chip Debugger–OpenOCD–,” *Fakultat fur Informatik, Tech. Rep*, 2006.
- [63] HP, “Pwn2own 2014: A recap; available at <http://www.pwn2own.com>,” 2014.

- [64] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.
- [65] T. INSTRUMENTS, “Stellaris LM3S2793 Microcontroller DATA SHEET,” 2014.
- [66] —, “LM3S2793 ROM USER’S GUIDE,” 2011.
- [67] —, “Pd9535 datasheet,” 2007. [Online]. Available: <http://www.ti.com/lit/ds/symlink/pca9535.pdf>
- [68] Intel, “Chapter 6 INTERRUPT AND EXCEPTION HANDLING,” in *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A: System Programming Guide*, 2020.
- [69] —, “Chapter 8 MULTIPLE-PROCESSOR MANAGEMENT,” *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide*, 2019.
- [70] —, “Chapter 4 PAGING,” in *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A: System Programming Guide*, 2020.
- [71] —, “Intel SGX Design Objectives,” 2013.
- [72] Intel Corporation, “Intel Software Guard Extensions (Intel SGX),” *ISCA 2015*, Jun 2015, reference No. 332680-002.
- [73] Intel Manual, “Chapter 1.5 ENCLAVE PAGE CACHE,” 2016.
- [74] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, “Preventing overflow attacks by memory randomization,” in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 339–347.
- [75] C. Jacobi, T. Slegel, and D. Greiner, “Transactional Memory Architecture and Implementation for IBM System Z,” in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. IEEE, 2012, pp. 25–36.
- [76] R. Joshi, H. Granada, and C. Tangpuz, “MOSFET BGA package,” in *2000 Proceedings. 50th Electronic Components and Technology Conference (Cat. No. 00CH37070)*. IEEE, 2000, pp. 944–947.
- [77] M. Jurczyk, “Bochspwn reloaded: Detecting kernel memory disclosure with x86 emulation and taint tracking,” *Black Hat USA*, 2017.
- [78] M. Jurczyk, G. Coldwind *et al.*, “Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns,” 2013.
- [79] M. Kaempf, “Smashing the Heap for Fun and Profit,” *Phrack Magazine*, vol. 11, no. 57, 2001.
- [80] D. Kedziorek and G. Czerniak, “HPC access using” KVM over IP”, in *2007 DoD High Performance Computing Modernization Program Users Group Conference*. IEEE, 2007, pp. 345–349.

- [81] G. Kestor, O. S. Unsal, A. Cristal, and S. Tasiran, “T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications,” in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 20.
- [82] B.-K. Kim, D.-H. Kang, J.-C. Na, and T.-M. Chung, “Abnormal traffic filtering mechanism for protecting ICS networks,” in *2016 18th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2016, pp. 436–440.
- [83] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, “Securing Real-Time Microcontroller Systems through Customized Memory View Switching,” in *NDSS*, 2018.
- [84] C. Konstantinou and M. Maniatakis, “Impact of firmware modification attacks on power systems field devices,” in *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2015, pp. 283–288.
- [85] P. Krishnan, “Hardware Breakpoint (or watchpoint) usage in Linux Kernel,” in *Proceedings of the Linux Symposium*. Citeseer, 2009, pp. 149–158.
- [86] I. Kuon, R. Tessier, and J. Rose, “Fpga architecture: Survey and challenges,” 2008.
- [87] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer Integrity,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 147–163.
- [88] R. Langner, “Stuxnet: Dissecting a cyberwarfare weapon,” *IEEE Security Privacy*, vol. 9, no. 3, pp. 49–51, 2011.
- [89] J. H. Lau, *Chip on Board: Technology for Multichip Modules*. Springer Science & Business Media, 1994.
- [90] K. Lawton, “Bochs: The Open Source IA-32 Emulation Project,” 2003.
- [91] B. Lee and J.-H. Lee, “Blockchain-based secure firmware update for embedded devices in an Internet of Things environment,” vol. 73, no. 3. Springer, 2017, pp. 1152–1167.
- [92] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, “Preventing Use-after-free with Dangling Pointers Nullification,” in *NDSS*, 2015.
- [93] R. P. Lee, K. Markantonakis, and R. N. Akram, “Binding hardware and software to prevent firmware modification and device counterfeiting,” in *Proceedings of the 2nd ACM international workshop on cyber-physical system security*, 2016, pp. 70–81.
- [94] F. Leens, “An introduction to I2C and SPI protocols,” *IEEE Instrumentation & Measurement Magazine*, vol. 12, no. 1, pp. 8–13, 2009.
- [95] Y. Li, J. M. McCune, and A. Perrig, “SBAP: Software-based attestation for peripherals,” in *International Conference on Trust and Trustworthy Computing*. Springer, 2010, pp. 16–29.

- [96] —, “Viper: verifying the integrity of peripherals’ firmware,” in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 3–16.
- [97] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics,” in *ACM Sigplan Notices*. ACM, 2008.
- [98] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.
- [99] D. Marino, M. Musuvathi, and S. Narayanasamy, “LiteRace: Effective Sampling for Lightweight Data-Race Detection,” in *ACM Sigplan Notices*. ACM, 2009.
- [100] B. Mark and E. Chris, “Significant flash exploit mitigations are live in v18.0.0.209; available at <https://googleprojectzero.blogspot.com>,” 2015.
- [101] Y. Mark, “Understanding IE’s New Exploit Mitigations: The Memory Protector and the Isolated Heap; available at <https://securityintelligence.com>,” 2014.
- [102] M. Matt, “Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP. <https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop>,” 2009.
- [103] J. McDonald and C. Valasek, “Practical Windows XP/2003 Heap Exploitation,” *Black Hat USA*, 2009.
- [104] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative Instructions and Software Model for Isolated Execution.” in *HASP@ ISCA*, 2013, p. 10.
- [105] L. McMinn and J. Butts, “A firmware verification tool for programmable logic controllers,” in *International Conference on Critical Infrastructure Protection*. Springer, 2012, pp. 59–69.
- [106] Microsoft, “Creating Guard Pages.”
- [107] —, “Microsoft Security Bulletin MS13-016; available at <https://docs.microsoft.com>,” 2013.
- [108] —, “Bug Check 0x133 DPC_WATCHDOG_VIOLATION; available at <https://docs.microsoft.com>,” 2017. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0x133-dpc-watchdog-violation>
- [109] —, “ProbeForRead Routine; available at <https://msdn.microsoft.com>,” 2017. [Online]. Available: [https://msdn.microsoft.com/en-us/library/windows/hardware/ff559876\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff559876(v=vs.85).aspx)

- [110] MITRE, “Common vulnerability exposure (CVE); available at <https://cve.mitre.org/>,” 2017.
- [111] H. Moore *et al.*, “The metasploit project; available at <https://www.metasploit.com/>,” 2009.
- [112] S. B. Moore, W. B. Glisson, and M. Yampolskiy, “Implications of malicious 3d printer firmware,” 2017.
- [113] B. Moran, M. Meriac, H. Tschofenig, and D. Brown, “A firmware update architecture for Internet of Things devices,” 2019.
- [114] Mozilla, “(CVE-2016-9079) Reported firefox SVG 0-day (Iterator invalidation in nsSMILTimeContainer::NotifyTimeChange()); available at <https://bugzilla.mozilla.org/>,” 2016.
- [115] —, “Firefox use-after-free in setbody; available at <https://www.mozilla.org/>,” 2013.
- [116] K. Murali, “Heap: Pleasures and Pains,” 1999.
- [117] MWR-Labs, “Isolated heap and friends - object allocation hardening in web browsers; available at <https://labs.mwrinfosecurity.com/>,” 2017.
- [118] S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Everything You Want to Know About Pointer-Based Checking,” in *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- [119] M. Neel, Riku, Antti, Matti, S. Jared, FiloSottile, M. Christian, Wvu, V. Juan, P. Sebastiano Di, S. Tom, Jjarmoc, B. Ben, and Herself, “MetaSploit Module: Openssl HeartBeat (HeartBleed) Information leak,” 2014.
- [120] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, “Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization,” *Intel Technology Journal*, vol. 10, no. 3, 2006.
- [121] N. Nethercote and J. Seward, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.
- [122] H. Neugass, G. Espin, H. Nunoe, R. Thomas, and D. Wilner, “VxWorks: an interactive development environment and real-time kernel for Gmicro,” in *Proceedings Eighth TRON Project Symposium*. IEEE, 1991, pp. 196–207.
- [123] T. Newman, T. Rad, L. ELCnetworks, J. Strauchs, and L. Strauchs, “Scada & plc vulnerabilities in correctional facilities,” *white paper*, pp. 1–14, 2011.
- [124] G. Novark and E. D. Berger, “DieHarder: securing the heap,” in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 573–584.
- [125] R. O’callahan and J.-D. Choi, “Hybrid dynamic data race detection,” in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2003, pp. 167–178.

- [126] J. Pan, G. Yan, and X. Fan, “Digtool: A virtualization-based framework for detecting kernel vulnerabilities,” in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.
- [127] B. Peter, “Substation attack is new evidence of grid vulnerabilit,” 2016.
- [128] E. Pozniarsky and A. Schuster, “MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.
- [129] V. Pravalika and R. Prasad, “Internet of things based home monitoring and device control using ESP32,” *International Journal of Recent Technology and Engineering*, vol. 8, no. 1S4, pp. 58–62, 2019.
- [130] Qualys, “GHOST: Glibc gethostbyname Buffer Overflow,” Qualys Security Advisory CVE-2015-0235, 2015.
- [131] N. A. Quynh and D. H. Vu, “Unicorn: Next generation cpu emulator framework,” *BlackHat USA*, 2015.
- [132] R. Rajwar and M. Dixon, “Intel Transactional Synchronization Extensions,” in *Intel Developer Forum San Francisco*, vol. 2012, 2012.
- [133] RAPID7, “Firefox xmlserializer use after free; available at <https://www.rapid7.com>,” 2013.
- [134] —, “Metasploit module mozilla_attribchildremoved; available at <https://www.rapid7.com>,” 2011.
- [135] W. K. Robertson, C. Kruegel, D. Mutz, and F. Valeur, “Run-time detection of heap-based overflows,” in *LISA*, vol. 3, 2003, pp. 51–60.
- [136] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.
- [137] K. Rosenfeld and R. Karri, “Attacks and Defenses for JTAG,” *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 36–47, 2010.
- [138] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A Dynamic Data Race Detector for Multithreaded Programs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.
- [139] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, “Protecting software through obfuscation: Can it keep pace with progress in code analysis?” *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–37, 2016.
- [140] M. Schulz, D. Wegemer, and M. Hollick, “Nexmon: Build your own wi-fi testbeds with low-level mac and phy-access using firmware patches on off-the-shelf mobile devices,” in *Proceedings of the 11th Workshop on Wireless Network Testbeds, Experimental Evaluation & CHaracterization*, 2017, pp. 59–66.
- [141] P. Semiconductors, “The i2c-bus specification,” *Philips Semiconductors*, vol. 9397, no. 750, p. 00954, 2000.

- [142] S. Sen, “Windows Debuggers: A WinDbg Tutorial,” 2004.
- [143] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A Fast Address Sanity Checker,” in *Presented as part of the 2012 USENIX Annual Technical Conference*, 2012, pp. 309–318.
- [144] F. J. Serna, “MS08-061: The Case of the Kernel Mode Double-Fetch; available at <https://msrc-blog.microsoft.com/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch>,” 2008.
- [145] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, “SWATT: Software-based attestation for embedded devices,” in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.* IEEE, 2004, pp. 272–282.
- [146] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and communications security.* ACM, 2007, pp. 552–561.
- [147] M. Shalan and V. J. Mooney, “A dynamic memory management unit for embedded real-time system-on-a-chip,” in *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, 2000, pp. 180–186.
- [148] N. Shavit and D. Touitou, “Software Transactional Memory,” *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.
- [149] T. Slaight, “Using IPMI platform management in modular computer systems,” 2003.
- [150] S. Soltan, M. Yannakakis, and G. Zussman, “Power grid state estimation following a joint cyber and physical attack,” vol. 5, no. 1. IEEE, 2016, pp. 499–512.
- [151] A. Sotirov, “Heap feng shui in javascript (2008),” 2008.
- [152] A. Sotirov and M. Dowd, “Bypassing Browser Memory Protections: Setting Back Browser Security by 10 Years.”
- [153] A. Sotirov, “Bypassing Memory Protections: The Future of Exploitation,” in *USENIX Security*, 2009.
- [154] A. Sotirov and M. Dowd, “Bypassing Browser Memory Protections in Windows Vista,” *Blackhat USA*, 2008.
- [155] Swiat, “Preventing the Exploitation of User Mode Heap Corruption Vulnerabilities,” 2008.
- [156] P. Team, “Pax address space layout randomization (aslr),” 2003.
- [157] uty and Saman, “How to hide a hook: A hypervisor for rootkits,” in *Phrack 69*, 2016.
- [158] C. Valasek, “Understanding the low fragmentation heap,” *Blackhat*, 2010.
- [159] C. Valasek and T. Mandt, “Windows 8 Heap Internals,” *Black Hat USA*, 2012.

- [160] N. Waisman, “Understanding and Bypassing Windows Heap Protection,” *Immunity*, 2007.
- [161] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, “How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei>
- [162] P. Wang, K. Lu, G. Li, and X. Zhou, “DFTracker: detecting double-fetch bugs by multi-taint parallel tracking,” *Frontiers of Computer Science*, vol. 13, no. 2, pp. 247–263, 2019.
- [163] X. Wang, C. Konstantinou, M. Maniatakis, and R. Karri, “Confirm: Detecting firmware modifications in embedded systems using hardware performance counters,” in *Proceedings of the IEEE/ACM international conference on computer-aided design*. IEEE Press, 2015, pp. 544–551.
- [164] R. N. Watson, “Exploiting Concurrency Vulnerabilities in System Call Wrappers,” *WOOT*, vol. 7, pp. 1–8, 2007.
- [165] J. Wei and C. Pu, “TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study,” in *FAST*, vol. 5, 2005, pp. 12–12.
- [166] W. William, “A fireside foray into a firefox fracas; available at <https://community.rapid7.com>,” 2016.
- [167] K. Williams and C. Bennett, “Why a power grid attack is a nightmare scenario,” vol. 30, 2016.
- [168] R. Wojtczuk and J. Rutkowska, “Attacking Intel Trusted Execution Technology,” *Black Hat DC*, p. 19, 2009.
- [169] J. Yang, A. Cui, S. J. Stolfo, and S. Sethumadhavan, “Concurrency Attacks,” *HotPar*, vol. 12, p. 15, 2012.
- [170] Y. Yang, “ROPs Are For The 99%.” 2014.
- [171] M. V. Yason, “WINDOWS 10 SEGMENT HEAP INTERNALS,” *Black Hat USA*, 2016.
- [172] —, “Understanding the attack surface and attack resilience of project spartan’s (edge) new edgehtml rendering engine,” 2015.
- [173] Y. Yu, T. Rodeheffer, and W. Chen, “Racetrack: Efficient detection of data race conditions via adaptive tracking,” in *ACM SIGOPS Operating Systems Review*. ACM, 2005.
- [174] M. Zeller, “Myth or reality—Does the aurora vulnerability pose a risk to my generator?” in *2011 64th Annual Conference for Protective Relay Engineers*. IEEE, 2011, pp. 130–136.

- [175] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, “IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time,” in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 71–86.
- [176] T. Zhang, D. Lee, and C. Jung, “TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory,” in *ACM SIGOPS Operating Systems Review*. ACM, 2016.
- [177] Z. Zhang, S. Gong, A. D. Dimitrovski, and H. Li, “Time synchronization attack in smart grid: Impact and analysis,” vol. 4, no. 1. IEEE, 2013, pp. 87–98.
- [178] L. Zhenhua, “Advanced Exploit Techniques Attacking the IE Script Engine.” Fortinet, Security Research, 2014.
- [179] —, “Advanced Heap Manipulation in Windows 8,” *Blackhat Europe*, 2013.