Too Subtle to Notice: Investigating Executable Stack Issues in Linux Systems

Hengkai Ye
The Pennsylvania State University
hengkai@psu.edu

Hong Hu
The Pennsylvania State University
honghu@psu.edu

***Abstract*—Code injection was a favored technique for attackers to exploit buffer overflow vulnerabilities decades ago. Subsequently, the widespread adoption of lightweight solutions like write-xor-execute (W⊕X) effectively mitigated most of these attacks by disallowing writable-and-executable memory. However, we observe multiple concerning cases where software developers accidentally disabled W⊕X and reintroduced executable stacks to popular applications. Although each violation has been properly fixed, a lingering question remains: what underlying factors contribute to these recurrent mistakes among developers, even in contemporary software development practices?**

In this paper, we conduct two investigations to gain a comprehensive understanding of the challenges associated with properly enforcing W⊕X in Linux systems. First, we delve into program-hardening tools to assess whether experienced security developers consistently catch the necessary steps to avoid executable stacks. Second, we analyze the enforcement of W⊕X on Linux by inspecting the source code of the compilation toolchain, the kernel, and the loader. Our investigation reveals that properly enforcing W⊕X on Linux requires close collaboration among multiple components. These tools form a complex chain of trust and dependency to safeguard the program stack. However, developers, including security researchers, may overlook the subtle yet essential `.note.GNU-stack` section when writing assembly code for various purposes, and inadvertently introduce executable stacks. For example, 11 program-hardening tools implemented as inlined reference monitors (IRM) introduce executable stacks to all "hardened" applications. Based on these findings, we discuss potential exploitation scenarios by attackers and provide suggestions to mitigate this issue.

I. Introduction

Code injection was a mainstream method for exploiting memory-safety issues (*e.g.*, buffer overflow) decades ago [70], [67], [16], [83], [57]. Attackers place malicious payloads, called *shellcode*, in controllable memory regions and corrupt control data (*e.g.*, return addresses and function pointers) to divert the control flow and execute the shellcode. At that time, stack was the ideal place to store shellcode, since attackers could leverage a single buffer overflow vulnerability to inject shellcode and corrupt the return address, both of which are on the stack.

Among all countermeasures against code-injection attacks, Write-XOR-Execute, also known as W⊕X, is one of the most straightforward and effective mechanisms [58], [60]. W⊕X allows every memory page to be either writable or executable, but not both at the same time. In this case, after an attacker inserts the malicious shellcode on the writable stack, the processor (CPU) will refuse to execute it and raise a segmentation fault. Thanks to its solid security benefit and negligible overhead, W⊕X is widely deployed in modern computers regardless of OSes, CPUs, and architectures. Attackers have to leverage more sophisticated code-reuse techniques [79], [19], [77] to achieve malicious goals, like remote code execution (RCE).

However, we have noticed multiple alarming cases where software developers accidentally disabled the W⊕X protection in their programs [14], [33], [20], [28], [8], [55], [13], [10], [15], [12], [34], [4], [3], [9], [11]. Victims include popular applications such as Electron [33], VSCode [14], and CockroachDB [34]. Several cases just happened in the past year. These cases are unexpected since W⊕X is a mature technique and should have been enabled automatically by default. Investigations revealed that W⊕X was disabled due to the inclusion of hand-written assembly files that missed a section directive: `.section .note.GNU-stack,"",@progbits`. This directive declares a `.note.GNU-stack` section to indicate the stack should not be executable. Since the meaning of this directive is not straightforward (some called it magic incantation [34]), developers failed to connect it to W⊕X. When manually writing assembly code for various reasons, they missed this directive and inadvertently made the application stack executable. In this paper, we term this problem BadAss.

Although these issues have been successfully fixed, we are curious about why developers occasionally miss this directive in hand-written assembly. Is it merely because general developers pay less attention to security features, or the implementation of W⊕X is too subtle for developers to catch? A comprehensive understanding of this problem is important for us to avoid similar security downgrades in the future. In this paper, we conduct investigations to reveal the underlying reason.

In the first step, we try to figure out whether experienced security researchers are more aware of this issue than general developers. If security researchers also make similar mistakes, we should not simply attribute the issue to general developers. We choose 21 popular inlined reference monitors (IRM) [41], [39], [40], [95] published at top-tier conferences or devel-

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230924
www.ndss-symposium.org

oped by recognized companies for investigation. We choose IRMs as our targets for three reasons. First, IRMs such as software-based fault isolation (SFI) [89], [59], [61], [94] are implemented by experienced security researchers, who have a comprehensive understanding of W$\oplus$X and a strong willingness to adopt security mechanisms. Second, compared with other security tools, IRMs are prone to BADASS issues since they commonly adopt hand-written assembly code to provide strong security guarantees. Third, since IRMs are developed to protect general applications, any implementation mistake may lead to catastrophic consequences to not a single, but many programs.

We use each IRM to transform a simple program and check the stack permission of the running process. To our surprise, 11 out of 21 tested IRMs disable W$\oplus$X, which means the hardened program will have an executable stack. This apparently violates the design goal of IRMs — regardless of security guarantees, hardening techniques should neither weaken nor disable existing defenses. The result indicates that the section-definition directive of W$\oplus$X is indeed too subtle. It is easy for developers, even experienced security researchers, to overlook it and make mistakes in hand-written assembly.

BADASS Issues in IRMs We identify BADASS issues from four CFI solutions, two in-process isolation techniques, and five binary reassemblers. MCFI [64], RockJIT [65], and πCFI [66] suffer from this issue since they include an empty assembly file in the program runtime. Since CFI prevents most illegal jumps to stack shellcode, BADASS makes the exploitation easier only if attackers can bypass the CFI, like through advanced ROP attacks [77]. PathArmor [88], ERIM [87], and Donky [76] introduce executable stacks due to dynamically loaded shared libraries with BADASS issues. PathArmor enforces CFI at the entries of sensitive syscalls; with BADASS, attackers can leverage stack shellcode to clean up invasion trajectory and help bypass CFI checks. ERIM adopts Intel MPK to provide lightweight in-memory isolation; with BADASS, attackers can inject and execute any MPK instructions to disable the isolation. Five binary reassemblers generate assembly files without `.note.GNU-stack` definitions. Unless users specify extra options to enable W$\oplus$X, the reassembled program will have an executable stack. We have reported our findings to the corresponding IRM developers. Ddisasm [42] fixed this issue before our report. Developers of the other 10 tools confirmed the executable stack issue. Authors of six tools have fixed this problem and another commits to patch the code.

W$\oplus$X Enforcement Analysis. In the second step, we investigate the W$\oplus$X implementation to understand and reveal its subtlety. We conduct a comprehensive inspection of Linux W$\oplus$X enforcement, especially in the GNU compilation toolchain – the default compiler on Linux systems. Our findings are as follows. ① If any C/C++ code contains address-taken nested functions that access local variables of enclosing functions [43], GCC will produce programs with executable stacks. At runtime, the program process will insert trampoline code on the stack and execute it to prepare arguments for nested functions [44]. ② If an assembly file misses the special directive, `.section .note.GNU-stack,"",@progbits`, the process will have an executable stack. ③ If an object file contains section `.note.GNU-stack` with flag `SEC_CODE`, or if it misses such a section, the process will have an executable stack. ④ If a binary contains segment `PT_GNU_STACK` with flag `PF_X`, or if it misses such a segment, the process will have an executable stack. Depending on the CPU, architecture and kernel version, all readable memory regions of the process could be executable due to the feature `READ_IMPLIES_EXEC` [75]. ⑤ If the `PT_GNU_STACK` segment of a shared library has the BADASS issue, the loader may make the process stack executable.

Based on our analysis, we find that most BADASS issues in popular applications and studied IRMs are caused by ② missing the definition of `.note.GNU-stack` in assembly files.

Generalization of BADASS. Our investigation shows that even experienced security researchers may overlook the subtle design of W$\oplus$X and introduce executable stacks. Inspired by this finding, we examine five additional practical security mechanisms to see whether they are also missed by security developers, specifically, PIE, RELRO, stack canary [32], FORTIFY_SOURCE [50] and Intel CET [69]. We adopt the same investigation methodology and get two new findings. First, three CFI solutions, MCFI, RockJIT and πCFI, accidentally disable FORTIFY_SOURCE, a security feature in GNU C library that detects a set of security vulnerabilities. Second, all IRM implementations that disable W$\oplus$X also turn Intel CET off. Similar to BADASS, the root cause is the inclusion of assembly files that lack section `.note.gnu.property`, which is required to enforce Intel CET. Creating `.note.gnu.property` requires a subtle assembly directive, `.section .note.gnu.property, "a"`, which is easy for developers to miss in hand-written assembly code.

In summary, we make the following contributions.

- We investigate the BADASS issue among popular inlined reference monitors (IRMs) [1] and find that many security researchers miss the subtle implementation of W$\oplus$X and introduce executable stacks to hardened programs.
- We conduct an in-depth analysis of W$\oplus$X enforcement on Linux systems, reveal the root cause of BADASS and demonstrate subtle code regions.
- We extend BADASS to other practical defenses and find that IRM implementations may accidentally weaken other techniques such as FORTIFY_SOURCE and Intel CET.

Roadmap. We organize the rest of the paper as follows. In §II, we introduce the background of code injection and W$\oplus$X and define our research problem. We present the overview of our research methodology in §III. We investigate the executable stack issue in program-hardening tools in §IV. We disclose the details of enforcing W$\oplus$X on Linux through code analysis in §V. We extend the stack issue and identify two more problems in §VI. In §VII, we illustrate the potential vector for attackers to deliberately introduce executable stacks, and discuss possible mitigation strategies. §VIII concludes the paper.

[1]Detailed instructions for the IRM investigation can be found at https://github.com/psu-security-universe/badass.

II. Background and Problem

In this section, we first introduce code-injection attacks and its countermeasure, W⊕X. Then, we define BadAss based on real-world cases and present our research problem.

A. Code Injection and W⊕X

Code injection is a well-known technique to exploit memory errors such as buffer overflow and use-after-free bugs [70], [67], [16], [83], [57]. To launch an attack, hackers first insert malicious shellcode into the memory space of a vulnerable program, and then divert the execution to the injected code. This technique was widely used in system hacking and cracking decades ago. For example, the notorious Morris Worm first injects a piece of VAXen shellcode on the stack of a vulnerable `fingerd` process. It then overwrites the return address to redirect the execution to the shellcode, which effectively starts a remote shell connection by invoking `execve("/bin/sh",0,0)` [81].

To prevent code-injection attacks, researchers proposed Write-XOR-Execute, or W⊕X [58], [60]. This property requires every memory page can either be writable or executable, but not both at the same time. With this protection, writable memory regions such as stack, heap, and .bss section are no longer executable. Even if attackers inject shellcode into such regions, attempts to execute the code will trigger a segmentation fault as the processor refuses to run these instructions. Enforcing W⊕X requires support from multiple components, including CPU and operating system (OS). The CPU reserves the most significant bit of the page table entry as the NX bit, and the OS sets the proper NX bit for each page. If the bit is zero, the CPU can execute code from that page. Otherwise, the page is non-executable. On the x86 architecture, AMD first implemented NX bit in 2003 and Intel followed later with a different name, XD bit. Mainstream operating systems such as Windows, Linux and OpenBSD started to support W⊕X.

Thanks to its strong security benefit and negligible overhead, modern OSes and mainstream processors have deployed W⊕X to prevent shellcode injection attacks. The adoption of W⊕X changed the rules of the in-memory war [83]. Attackers had to seek new opportunities from existing code sections and invented a set of code-reuse methods to achieve arbitrary code execution, like return-to-libc (ret2libc) [63] and return-oriented programming (ROP) [79], [19], [77], [23]. However, code-reuse attacks are more complicated than code injection. For example, to use the method of ROP, attackers must orchestrate a large number of small code gadgets to construct meaningful payloads [78]. To minimize the challenge, modern exploitations commonly start with a few gadgets to disable W⊕X and create writable-and-executable pages, and then utilize code injection to achieve final goals [83], [97]. Security researchers also moved their attention to prevent various code-reuse attacks, like through address randomization [71], control-flow integrity (CFI) [1], [21] and code-pointer integrity (CPI) [53], [56].

B. Problem Definition

While W⊕X is a mature defense technique and has been deployed extensively, we have observed numerous concerning instances where software developers inadvertently deactivated the W⊕X protection within their programs [14], [33], [20], [28], [8], [55], [13], [10], [15], [12], [34], [4], [3], [9], [11], [62], [5]. Such unexpected security downgrades have been noted even within widely used applications, including those of recent occurrence in the year 2023. For example, CockroachDB, a popular commercial distributed database management system (DBMS) [86], used to contain an executable stack due to accidentally introducing an assembly file [34]. Electron, a software framework to build cross-platform desktop applications, also reported the executable stack issue [33]. Many downstream programs developed on top of Electron get affected, like Visual Studio Code, Mattermost, Wire Desktop, and Rocket Chat. By reviewing discussions on these reports, we identify that hand-written assembly files without the section-definition directive `.section .note.GNU-stack,"",@progbits` are the culprit. These discussions also mention that this problem stems from historical design choices for compatibility purposes and resides in the large complicated code base. In this paper, we term this problem BadAss.

Definition II.1 (BadAss)**.** On Linux systems, if the code of an application contains any assembly file that misses the section-definition directive `.section .note.GNU-stack,"",@progbits`, the application binary generated by the GNU compilation toolchain will have an executable stack. Specifically, when we load such a binary into memory for execution, W⊕X will be turned off and the process stack will be both *writable and executable*. In the worst case, all readable memory regions of the process will also be executable, including writable heap, .bss and .data sections. We refer to this issue as BadAss.

The writable-and-executable memory region introduced by BadAss will streamline the bug-exploitation process. Attackers can directly inject malicious code into the vulnerable process like decades ago [70], [67], [16], [57]. Although they still need to bypass protections in other phases, BadAss eliminates the need for laborious code-reuse attacks that involve tedious searching and chaining of code gadgets [79], [19], [77].

Figure 1 demonstrates the BadAss issue. File hello_world.c is a simple C program that merely waits for a character from the command line and exits immediately once the input is received. File a.s is an empty assembly file without any malicious code. When we compile hello_world.c alone through GCC, the generated binary hello_world has a readable and writable stack, but not executable, shown as `rw-` from the process memory mapping. However, when we include the empty assembly file a.s into the compilation process, the generated binary will have a readable, writable and executable stack, shown as `rwx` in the memory mapping. In fact, assembly instructions inside a.s do not matter. The order of compilation also makes no difference. The generated binary will have an executable stack unless the assembly file a.s explicitly includes the following section-definition directive: `.section .note.GNU-stack,"",@progbits`.

Research Problem. BadAss is not entirely obscure to the

```
$ cat hello_world.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  getchar();
  return 0;
}
$ # a.s is an empty assembly file
$ cat a.s

$ ###################################### compile using gcc
$ gcc hello_world.c -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffec4016000-7ffec4037000 rw-p 00000000 00:00 0    [stack]
$ #                       ^^   readable and writable

$ kill $(pgrep hello_world)
$ ############################# compile with a.s using gcc
$ gcc hello_world.c a.s -o hello_world
$ ./hello_world
^Z[1] + Stopped                    ./hello_world
$ cat /proc/$(pgrep hello_world)/maps | grep stack
7ffd8d5a0000-7ffd8d5c1000 rwxp 00000000 00:00 0    [stack]
$ #                       ^^^  readable, writable and executable
```

Fig. 1: Demonstration of BADASS. When compiled with an empty assembly file a.s, the process has an executable stack.

public, as it has been disclosed and discussed several times within small groups [54], [84], [6], [7], [92]. Nonetheless, our focus lies in understanding *why developers occasionally overlook it and introduce security downgrades even in widely used applications, including recent occurrences*. Understanding the root cause is both intriguing and imperative for us to take proper actions and avoid similar issues in the future. We propose conjectures with two potential reasons.

- *Developer unawareness.* One potential explanation is that most developers primely concentrate on program functionalities and often pay limited attention to even fundamental security features such as W⊕X. For example, one CockroachDB developer created an empty assembly file simply to make the optimization work [34]. Although the file introduced an executable stack, the corresponding commit passed the code review, was merged into the code base smoothly, and lurked for four months. Another developer caught this problem only when the application failed to run on SELinux due to the policy `execmem`, which prohibited mapping a region of memory with both `PROT_WRITE` and `PROT_EXEC` [34]. If this is the main reason, we should train normal developers to be more aware of security features.
- *Subtlety in W⊕X enforcement.* Another possibility is that the design and implementation of W⊕X make it easy for developers to overlook it, especially in hand-written assembly code. Even if developers keep security features in mind and follow all default system settings, the compiled binary will have an executable stack. If this is the root cause, we may need to rethink the enforcement of W⊕X to improve the program security in default settings.

In this paper, we will investigate the BADASS issue to figure out the main root cause. Our goal is to help alarm developers and the community to avoid similar issues in the future.

III. OVERVIEW

Now we have two ways to explain why BADASS happens frequently and recently in popular applications. Based on our understanding of the problem, we design two comprehensive investigations to identify the main root cause out of these two.

Our first investigation aims to verify that general software developers primarily focus on program functionalities and lack security awareness. However, measuring the security awareness of general developers is a challenging task and may produce subjective results. Therefore, we try to verify this conjecture from a different perspective by answering another question, *i.e.*, whether experienced security developers also miss the obscure assembly directive and introduce executable stacks to security applications. Our observation is that security developers usually possess a thorough understanding of security mechanisms, pay special attention to security features, and demonstrate a strong willingness to strengthen application security rather than compromise it. If they also overlook the BADASS issue, we cannot simply attribute this problem to developer's unawareness of security features. We will explain the investigation methodology, present our findings and discuss the security implications in §IV.

The second investigation is to understand whether the enforcement of W⊕X contains subtle design and implementation details that are challenging for developers to notice. While missing the section-definition directive seems to be a common reason for BADASS, we are unclear how this assembly section affects the executable permission of the process stack. In §V, we will follow the program lifetime to investigate the GNU compilation toolchain and the Linux kernel to reveal the subtlety within the W⊕X enforcement. To the best of our knowledge, this is the first in-depth investigation of this issue.

IV. INVESTIGATING SECURITY TOOLS

In this section, we plan to investigate the BADASS issue in security applications developed by experienced security programmers or researchers. However, security tools encompass a wide range of software and hardware solutions to protect systems, networks, and even data. To reduce the scope of analysis, we define three criteria to help identify security applications that are most related to BADASS. It is worth mentioning that our investigation aims to identify concrete cases where security developers miss the critical assembly line and introduce executable stacks. Therefore, we do not have to cover all security productions. ① The tool should utilize assembly code for various purposes, such as enhancing security or improving performance. Based on our observation of previous BADASS issues, incorrect handling of assembly code is the main reason for introducing executable stacks. ② The tool should interact with native ELF binaries, like generating new binaries or modifying existing ones. ③ It will be good if the tool is open-sourced and works on Linux. This property will make our investigation easier. After examining various types of security tools, we find that inlined reference monitor (IRM) aligns with our expectations [41], [39], [40], [95]. Next, we first provide a brief introduction to IRMs and

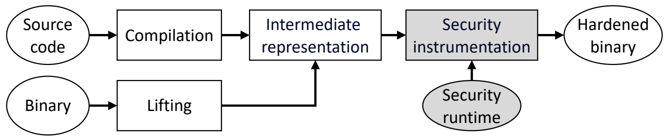


Fig. 2: Overview of IRM implementation

explain why they are ideal targets for BADASS investigation (§IV-A); then, we introduce our methodology for investigating IRMs (§IV-B); at last, we present our findings and discuss the security implications (§IV-C).

A. Inlined Reference Monitor (IRM)

Inlined reference monitor (IRM) generally injects security checks into subject programs to enforce security properties [41], [39], [40], [95]. It has been widely adopted to enforce various security policies, especially to block sophisticated and powerful in-memory attacks [79], [19], [77], [49]. For example, IRMs are used to enforce control-flow integrity (CFI) [64], [65], [66], [88], a principled solution to prevent control-flow hijacking attacks [21], [1], [96]. First, CFI solutions take various static or dynamic analyses to infer legal targets for each indirect call, jump and return instruction (or *icall* for short). Second, at runtime, CFI checks whether each icall instruction jumps to one of the inferred legal targets; if not, it reports the CFI violation. Researchers commonly adopt IRM in the second step, which inserts security checks before icall instructions to compare with predefined targets. Software-based fault isolation (SFI) is another security policy commonly implemented using IRM [89], [59], [61], [94]. It splits the memory space into multiple trusted or untrusted domains, and inserts security checks before every memory-access or icall instruction to ensure that the access is confined to the corresponding domain.

Figure 2 illustrates two general methods for IRM to harden programs. If the source code is available, we can utilize the compiler to insert checks during compilation. Particularly, we use compiler analysis to generate informative metadata, like data flow, and leverage these metadata to insert necessary checks into proper code locations. If the source code is unavailable, we have to inject checks through binary-rewriting techniques [72], [68]. In this case, we usually first lift program binaries to a higher-level representation, like assembly [91], [36], [37], [42], customized IR [93], or source code [17]. Then, we insert security checks into proper locations and compile the instrumented code into hardened binaries. Regardless of the method, the injected code will perform security checks and report detected errors during the program execution. Other than security instrumentation, IRMs commonly adopt special runtime to assist with checks and reports. A runtime is usually developed in native languages (*e.g.*, C/C++) or assembly, and gets compiled into the object file or shared library.

Ideal Targets for BADASS Investigation. IRMs are the ideal targets of BADASS investigation for three reasons. ① IRMs, such as software-based fault isolation (SFI) and control-flow integrity (CFI), are designed and implemented by experienced security researchers or well-known commercial companies to bolster program security. Compared to general developers, security researchers are anticipated to possess a more thorough understanding of W$\oplus$X and are more sensitive to security violations like disabling W$\oplus$X. Therefore, if we can find BADASS in IRMs, that will be a strong demonstration of the challenge to notice BADASS. ② Based on existing BADASS issues, we speculate the usage of assembly code is positively correlated with the introduction of BADASS. IRMs commonly adopt hand-written assembly code for various purposes, like providing a strong security guarantee [56], lifting a binary into an intermediate representation [91], [37], [80] or implementing a runtime for instrumentation [64], [66], [65]. For example, ERIM, an MPK-based isolation technique, uses an assembly file to hinder untrusted components from creating executable memory [87]. ③ IRMs are designed to protect general applications. If an IRM suffers from BADASS, any hardened applications may also be affected. This situation is more severe than BADASS in a single application.

B. Methodology

We collect 21 open-source IRMs implemented by recognized security researchers and developers, including nine binary rewriters, eight CFI solutions, two in-process isolation methods, one SFI solution, and one binary debloating tool, as shown in Table I. Most developers of these tested IRMs have published their papers at top-tier academic conferences on security, system, or programming language. 17 tools are proposed by academic research groups and the other 4 tools, specifically, LLVM CFI [85], Android kCFI, Wasmtime and Ddisasm [42] are developed by well-known security companies such as Google, GrammaTech, and Bytecode Alliance. Android kCFI is implemented based on LLVM CFI and has been enabled by all Android devices since 2018. Wasmtime is a popular runtime for WebAssembly and has deployed various sandboxing strategies. Another ever-popular IRM is NaCl proposed by Google [94]. However, it has been deprecated for years and we did not find a good method to build and run it on modern systems. Four IRMs, binCFI [98], ERIM [87], Ramblr [90] and Ddisasm, received Distinguished Paper Awards from top-tier security conferences. ERIM was also awarded the Facebook Internet Defense Prize. Uroboros [91], a reassembleable disassembler, is adopted by two teams among seven finalists during the DARPA Cyber Grand Challenge (CGC) competition [35]. Ramblr is developed and also used in CGC by team Shellphish and has been integrated into angr [80]. MCFI, RockJIT, and πCFI are used in Google CTF 2017.

Our investigation starts with a simple program. For IRMs that work on program source code, we follow tutorials on their official websites to compile the program and obtain a hardened binary. If the IRM works on binaries, we first prepare a binary with W$\oplus$X enabled and apply the IRM to get a hardened version. After obtaining the hardened binary, we run the program and check the in-memory stack permission. If the hardened process has any writable-and-executable memory region while the original version does not, we consider the corresponding IRM suffers from the BADASS issue. We then inspect the IRM code to figure out the defective assembly file.

TABLE I: Inlined reference monitors for investigation. For LLVM CFI and Wasmtime, the link is for the whole project repository. binCFI and SAFER are released on their own websites (not Github). 🏆: Distinguished Paper Award; 🛡: Facebook Internet Defense Prize

Category	IRM	Version	Ref.	Conference	Year	Award	Citation	Stars	Forks	Assembly	BADASS?	Status
Control Flow Integrity (CFI)	binCFI ↗	-	[98]	USENIX SEC	2013	🏆	775	-	-	yes	no	-
	MCFI ↗	7c9de0c	[64]	PLDI	2014		313	41	7	yes	yes	fixed
	LLVM CFI ↗	11	[85]	USENIX SEC	2014		542	26.9k	11k	no	no	-
	RockJIT ↗	7c9de0c	[65]	CCS	2014		165	41	7	yes	yes	fixed
	πCFI ↗	7c9de0c	[66]	CCS	2015		234	41	7	yes	yes	fixed
	PathArmor ↗	9879a85	[88]	CCS	2015		334	45	15	yes	yes	won't fix
	μCFI ↗	87cfea3	[48]	CCS	2018		137	17	4	yes	no	-
	Android kCFI ↗	16-6.1	-	-	2018		-	-	-	no	no	-
SFI	Wasmtime ↗	ab2aae5	-	-	2017		-	14.8k	1.2k	no	no	-
Debloating	Razor ↗	c61403f	[74]	USENIX SEC	2019		121	39	8	yes	no	-
Isolation	ERIM ↗	f1d4a28	[87]	USENIX SEC	2019	🏆🛡	250	13	6	yes	yes	fixed
	Donky ↗	d97ca2a	[76]	USENIX SEC	2020		93	14	5	yes	yes	no risk
Binary Rewriting	Uroboros ↗	c074cca	[91]	USENIX SEC	2015		168	187	56	yes	yes	checking
	Ramblr ↗	1ef5509	[90]	NDSS	2017	🏆	179	249	46	yes	yes	will fix
	Multiverse ↗	1808198	[18]	NDSS	2018		136	298	33	no	no	-
	Egalito ↗	c5bccb4	[93]	ASPLOS	2020		101	207	37	no	no	-
	RetroWrite ↗	d722ec5	[37]	IEEE S&P	2020		174	655	78	yes	yes	fixed
	E9Patch ↗	5bb07ff	[38]	PLDI	2020		75	937	65	no	no	-
	Ddisasm ↗	1.5.4	[42]	USENIX SEC	2020	🏆	89	637	58	yes	yes	fixed
		1.8.0									no	-
	ARMore ↗	d722ec5	[36]	USENIX SEC	2023		4	655	78	yes	yes	fixed
	SAFER ↗	-	[73]	USENIX SEC	2023		1	-	-	yes	no	-

C. Result and Implication

Our investigation reveals unexpected results, where 11 out of 21 IRMs have the BADASS issue and introduce executable stacks to hardened applications. We find the problems across three categories, including four CFI solutions (MCFI, RockJIT, πCFI and PathArmor), two in-process isolation methods (ERIM and Donky), and five binary rewriters (Uroboros, Ramblr, RetroWrite, Ddisasm and ARMore). In the category of binary rewriting tools, all rewriters decompile binary code into reassembleable files, but none of them contains the essential `.note.GNU-stack` section to disable executable stacks. This result shows that even experienced security researchers also miss the subtle design of W$\oplus$X and introduce BADASS issues. Therefore, our first conjecture is likely invalid — we cannot simply attribute existing BADASS issues to the unawareness of security features among general developers.

We also assess the impact of an executable stack on the hardened program in the context of the proposed security policy, such as CFI and SFI. MCFI/RockJIT/πCFI are three CFI solutions that require each icall instruction to jump to one predefined legitimate location. Since in most cases, the process stack region does not contain any valid code, no stack address is a valid target for any icall. Only if attackers already have bypassed MCFI/RockJIT/πCFI using advanced exploitation methods, like control-flow bending [22] or counterfeit object-oriented programming [77], BADASS will make the subsequent exploitation easier. Therefore, the security impact of BADASS on MCFI/RockJIT/πCFI is limited. Nevertheless, developers of these tools have fixed this issue immediately after our report.

PathArmor is a practical context-sensitive CFI solution [88]. At the entry of security-sensitive syscalls (*e.g.*, `execve` that executes a new program), it will pause the execution and verify the last 16 icalls are all valid. It utilizes hardware features to accelerate branch recording and verification. Since PathArmor only checks the last 16 icalls before each syscall, attackers may bypass the protection by building a long gadget chain that contains at least 16 valid icalls at the end. This is usually believed challenging since attackers have to guarantee memory accesses within 16 icalls are all legitimate to avoid crashes. However, with an executable stack, such advanced attacks are more practical. In particular, attackers can use a memory error to jump to the shellcode on the stack, where they can execute any instructions to update the memory content properly. As long as these instructions do not trigger security-critical syscalls, PathArmor will not conduct any security checks. After manipulating the memory to the expected content, attackers can invoke the long gadget, where all memory accesses within the last 16 icalls are guaranteed legitimate and the execution will reach critical syscalls. Since the last 16 icalls are all legitimate, PathArmor will let the execution continue. In this scenario, the executable stack provides an effective method to bypass PathArmor, reducing the security of the hardened program. PathArmor developers confirmed the executable stack and the security risk. However, they decide not to fix the problem since the code is not maintained by anyone anymore.

ERIM and Donky are two intra-process isolation mechanisms [87], [76]. ERIM utilizes the hardware feature, Intel memory protection keys (MPK), to achieve efficient permission control. As part of the isolation enforcement, ERIM statically identifies and eliminates all unexpected MPK-related instructions from the code section, and dynamically prevents untrusted components from creating executable memory regions. However, with an executable stack, attackers can inject any shellcode on the stack, including all MPK-related instructions. They can update the permission settings at runtime, and effectively break the proposed in-process isolation. ERIM

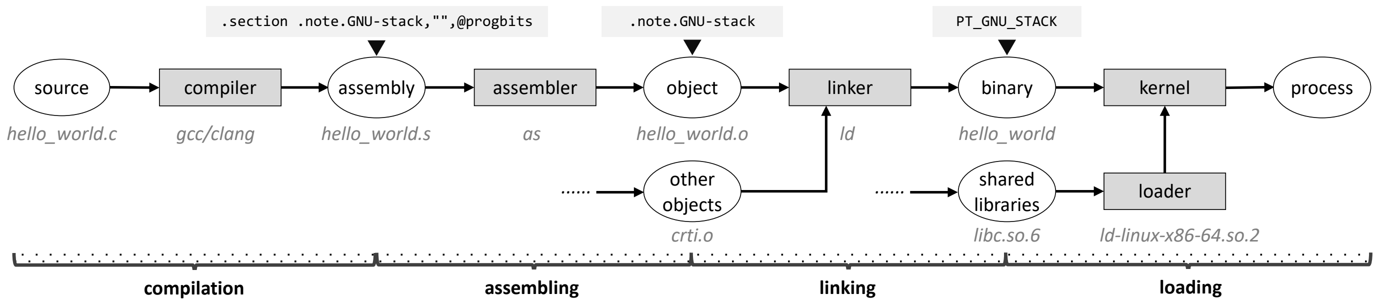


Fig. 3: Overview of program lifetime. The source code is first compiled into an assembly code, which is then assembled to an object file. The object file is linked with other objects to generate a binary, which is finally loaded into the process with shared libraries.

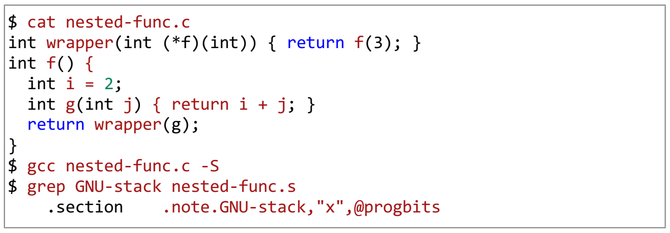
developers have fixed this issue based on our report. Although Donky provides similar isolations, BADASS has a limited security impact on it. This is because Donky will create a separated non-executable stack for each isolated domain, while the default stack which is executable due to BADASS will not be used by any code. Thanks to the isolation, code in other domains cannot jump to the default executable stack.

All five binary rewriters, including Uroboros, Ramblr, RetroWrite, Ddisasm and ARMore are reassembleable disassemblers, or reassemblers for short. A reassembler first disassembles the binary code into valid assembly files. After optional instrumentation, like adding memory-safety checks, it will assemble the assembly file back into binaries. Unlike previous IRMs which enforce specific security properties, binary rewriter is a general platform to help insert various security properties to binaries. Therefore, the security implication of BADASS also varies. For example, RetroWrite is mainly designed for fuzzing-oriented instrumentation and sanitization, where attackers have no chance to reach the modified binary. However, we still suggest fixing this issue to eliminate any opportunity that weakens the application security. These tools could be used for security purposes in the future, like how Uroboros and Ramblr are adopted in DARPA CGC [35]. Ddisasm developers fixed this issue in November 2022, long before we started our investigation. Developers of RetroWrite and ARMore have fixed BADASS based on our reports, and Ramblr developers have committed to fixing it in the near future. Uroboros developers are still investigating the problem.

Takeaway: Even experienced security researchers and recognized developers may miss the section-definition directive for W⊕X when developing security-hardening tools. Applications protected by these security tools will have unexpected executable stacks, which downgrade the program security to a certain extent. We cannot simply attribute all BADASS problems to the unawareness of security features among general developers.

V. ANALYZING W⊕X ENFORCEMENT

We proceed to check our second conjecture - *the obscure assembly directive is too subtle to catch and the W⊕X enforcement is complex and hard to follow*. To achieve our goal, we conduct an in-depth analysis to comprehensively understand how W⊕X is enforced on Linux. Our analysis follows the program lifetime shown in Figure 3, spanning the source code of the GNU compiler (§V-A), assembler (§V-B), linker (§V-C), Linux kernel (§V-D) and loader (§V-E). We aim to reveal various assumptions and dependencies among multiple components to enforce a non-executable stack (§V-F). To avoid BADASS, we must ensure all assumptions and dependencies are satisfied during the program compilation and process creation. Along with the analysis, we will discuss how investigated IRMs introduce executable stacks to protected applications.



```
$ cat nested-func.c
int wrapper(int (*f)(int)) { return f(3); }
int f() {
  int i = 2;
  int g(int j) { return i + j; }
  return wrapper(g);
}
$ gcc nested-func.c -S
$ grep GNU-stack nested-func.s
    .section    .note.GNU-stack,"x",@progbits
```

Fig. 4: Nested functions leading to executable stack. GCC will create an executable `.note.GNU-stack` section in the assembly.

A. Compilation

Compilation is the process that translates the program from source code to assembly. During compilation, the GNU compiler GCC will *always* insert the section-definition directive `.section .note.GNU-stack,"`*`flags`*`",@progbits` into the assembly. This directive defines a `.note.GNU-stack` section to indicate whether the stack should be executable. The default value of *`flags`* is an empty string, and the stack will be non-executable. But in rare cases, the source code may define a special nested function (*i.e.*, defining a new function within another) that accesses local variables of the enclosing function and its address is explicitly taken [43]. For programs with such special nested functions, GCC will generate specialized binaries with executable stacks. These binaries will dynamically insert specific trampoline code onto the stack during runtime, and execute these code to prepare arguments for the nested functions [44]. In this case, GCC will set *`flags`* to "x" to indicate that the stack has to be executable.

Figure 4 shows an example where special nested functions lead to an executable stack, inspired by the blog [84]. Function `g` is defined within function `f`. It accesses the local variable `i` of `f`, and its function address is taken and passed as an argument to function `wrapper`. After we compile nested-func.c with GCC to assembly, we can find "x" is used in the *`flags`* of section `.note.GNU-stack`. Once `nested-func.c` is compiled

TABLE II: Compilation result. GCC always emits the directive for `.note.GNU-stack`, and sets the flags based on nested functions.

IN: one source code	OUT: one assembly code
define any nested function that • accesses local variables of the enclosing function • has function address explicitly taken	↪ `.section .note.GNU-stack,"x",@progbits`
otherwise	↪ `.section .note.GNU-stack,"",@progbits`

```
void file_end_indicate_exec_stack(void) {
  unsigned int flags = SECTION_DEBUG;
  if (trampolines_created)  // 1 when creating trampoline
    flags |= SECTION_CODE;
  switch_to_section(get_section(".note.GNU-stack",flags,0)); }
```

into an application, the process will have an executable stack, as shown in Figure 1. The bottom of Table II shows the related GCC code. It creates `.note.GNU-stack` and sets flags based on variable `trampolines_created`. GCC sets this variable to `true` once it creates any trampoline code for special nested functions. We summarize the behavior of GCC in Table II.

Nested function is not supported in standard C and C++, and is merely an extension to GNU C (even not allowed in GNU C++). Although such functions were ever used in particular applications before, such as grub [8], link-grammar [26] and mountall [27], developers have rewritten their applications to eliminate nested functions. GCC supports this feature merely for compatibility purposes, while other compilers do not support it. For example, CLANG does not allow nested functions and always uses empty `flags` in the directive to create non-executable `.note.GNU-stack` sections. None of our investigated security tools in §IV adopt nested functions. In summary, as long as we compile an application from the source code, the emitted assembly files will always contain the section-definition directive, which in most cases indicates non-executable stacks.

B. Assembling

GNU assembler AS maps every assembly instruction to machine code, and creates code and data sections based on various directives. Developers can provide command-line options `--execstack` and `--noexecstack` to force AS to generate the `.note.GNU-stack` section and set proper executable permission (lines 2-7 and 15-18 of the code in Table III). However, if the developers do not provide such options, AS checks the assembly file for creating required sections. If the assembly file contains the `.section .note.GNU-stack,"`*`flags`*`",@progbits` directive, AS will create an empty `.note.GNU-stack` section (lines 9 and 12). If *`flags`* is "x", AS will set the `SEC_CODE` flag in the section header (lines 11 and 13), indicating the stack should be executable. Otherwise, the header has no special flags. However, if the assembly file does not have any directive for `.note.GNU-stack` and the developers also do not specify any option, the assembler will <u>*not*</u> create this section in the ELF object file. Similarly, another popular assembler, NASM, checks the NASM syntax assmebly file for an equivalent assembly directive `section .note.GNU-stack noalloc noexec nowrite progbits` and handles `.note.GNU-stack` just like AS. We

TABLE III: Assembling result. AS will translate assembly files to machine code verbatim, and create `.note.GNU-stack` accordingly.

IN: one assembly code	OUT: one object code
`.section .note.GNU-stack,"x",@progbits` | `--execstack`	↪ `.note.GNU-stack` w/ SEC_CODE
`.section .note.GNU-stack,"",@progbits` | `--noexecstack`	↪ `.note.GNU-stack` w/o SEC_CODE
directive missing	↪ `.note.GNU-stack` missing

```
1  // 1. ---- Parse command-line arguments ------------------
2  int optc = getopt_long_only(...);
3  if (optc == OPTION_EXECSTACK) {
4    flag_execstack = 1;   flag_noexecstack = 0;
5  } else if (optc == OPTION_NOEXECSTACK) {
6    flag_execstack = 0;   flag_noexecstack = 1;
7  }
8  // 2. ---- Create sections based on directives -----------
9  char *name = obj_elf_section_name();     // .note.GNU-stack
10 bfd_vma attr = obj_elf_parse_section_letters(...);  // "x"
11 flagword flags = ((attr & SHF_EXECINSTR) ? SEC_CODE : 0);
12 segT sec = subseg_force_new (name, 0);
13 bfd_set_section_flags (sec, flags);
14 // 3. ---- Create/Update based on command-line arguments -
15 if ((flag_execstack || flag_noexecstack) && ...) {
16   segT gnustack = subseg_new (".note.GNU-stack", 0);
17   bfd_set_section_flags (gnustack,
18           SEC_READONLY | (flag_execstack ? SEC_CODE : 0)); }
```

summarize three different settings of the `.note.GNU-stack` section in an ELF object file in Table III.

Based on the discussion in §V-A, if the assembly file is produced by a compiler, like GCC or CLANG, it must contain a directive for `.note.GNU-stack`. In that case, the assembler will always create the `.note.GNU-stack` section. However, not all code is written in high-level languages and follows the complete compilation chain in Figure 3. Hand-written assembly files intervene in the compilation process and the W⊕X enforcement path from the middle. As a consequence, they are not processed by the compiler and may not have this directive. Our investigation in §IV-C reveals that all 11 affected IRMs contain at least one assembly file missing the obscure directive. The assembly code may come from hand-written assembly files (*e.g.*, ERIM), empty assembly files generated from building scripts (*e.g.*, MCFI), or assembly files disassembled from binary (*e.g.*, RetroWrite). Following the last case in Table III, with these assembly files, AS will create object files without `.note.GNU-stack` sections.

C. Linking

GNU linker LD combines all object files and produces ELF binaries, including executables (*e.g.*, hello_world) and shared libraries (*e.g.*, libc.so). During linking, LD drops all `.note.GNU-stack` sections in ELF object files and creates a `PT_GNU_STACK` segment header to indicate whether the stack should be executable. Table IV summarizes the related code for this task. First, LD supports two command-line options `-z execstack` and `-z noexecstack` for developers to explicitly specify the stack permission. Once developers use such options, LD will create a `PT_GNU_STACK` segment and set the expected permission. Lines 2-8, 10-13 and 32-35 show the logic. If developers specify multiple `-z` options, the last one will overwrite all previous settings. At lines 11 and 13, LD will change `elf_stack_flags` from the value 0 to readable (`PF_R`),

TABLE IV: Linking result. Based on whether object files contain `.note.GNU-stack` sections and whether such section has "x" flag, LD will create the `PT_GNU_STACK` segment and set executable permission.

IN: multiple object files		**OUT: one executable**	
`.note.GNU-stack`	`SEC_CODE`	`PT_GNU_STACK`	`PF_X`
all have	① any	created	yes
	② none	created	no
all miss	③ -	missing	-
some have, others miss	① any	created	yes
	④ none	created	yes (x86/x64/arm) no (aarch64)

```
1  //----- 1. Parse command-line arguments: -z optarg ----------
2  if (strcmp(optarg, "execstack") == 0) {
3    link_info.execstack = true;
4    link_info.noexecstack = false;
5  } else if (strcmp(optarg, "noexecstack") == 0) {
6    link_info.noexecstack = true;
7    link_info.execstack = false;
8  }
9  //----- 2. Set elf_stack_flags -----------------------------
10 if (info->execstack)                    // info = &link_info;
11   elf_stack_flags = PF_R | PF_W | PF_X;
12 else if (info->noexecstack)
13   elf_stack_flags = PF_R | PF_W;
14 else {
15   asection *notesec = NULL, *s = NULL; int exec = 0;
16   // Traverse every object file
17   for (obj = info->input_bfds; obj; obj = obj->link.next) {
18     s = bfd_get_section_by_name(obj, ".note.GNU-stack");
19     if (s) {                           // has .note.GNU-stack
20       notesec = s;
21       if (s->flags & SEC_CODE) {             // "x" in flags
22         exec = PF_X; break;
23     } }
24     else if ( bed->default_execstack &&
25               info->default_execstack)
26       exec = PF_X;                      // no .note.GNU-stack
27   }                            // end of object-file traversal
28   if (notesec || info->stacksize > 0)
29     elf_stack_flags = PF_R | PF_W | exec;
30 }
31 //----- 3. Create PT_GNU_STACK segment ----------------------
32 if (elf_stack_flags) {
33   m = bfd_zalloc (abfd, ...);
34   m->p_type = PT_GNU_STACK;
35   m->p_flags = elf_stack_flags;      }
```

writable (`PF_W`) and if necessary, executable (`PF_X`). As long as `elf_stack_flags` is not 0, lines 32-35 will create the segment with the proper type and permission.

However, not all developers specify stack permissions in linking flags. In this case, LD will leverage the `.note.GNU-stack` section in *every* object file to determine the stack permission. At line 15, it allocates `notesec` to record whether any object file has a `.note.GNU-stack` and uses `exec` to accumulate the stack permission. Lines from 17 to 27 form a loop to check all provided object files one by one. For every object file, LD searches for `.note.GNU-stack` from section headers. ① If it can find this section, LD records the search result in `notesec`. Further, if this section in the current object file requires executable permission, the code records it in `exec` and exits the loop immediately. Following the early exit, since `notesec` is not `NULL`, LD will update `elf_stack_flags` at line 29 (`exec` is `PF_X`), and create the `PT_GNU_STACK` segment as executable. ② If all object files have `.note.GNU-stack` sections but none of them are required to be executable, `exec` will be always 0 and LD will create a non-executable `PT_GNU_STACK`

TABLE V: Kernel loading result and related code, after merging the rules of `READ_IMPLIES_EXEC` (details provided in Table VI).

IN: one binary	**OUT: one process**			
`PT_GNU_STACK`	arm	kernel	kernel >= v5.8-rc1	
	< v6	< v5.8-rc1	x86/>=armv6	x64/aarch64
`PF_R|PF_W`	exec-all	no	no	no
`PF_R|PF_W|PF_X`	*exec-all*	*exec-all*	exec-stack	exec-stack
segment missing	exec-all	exec-all	exec-all	no

```
//---- 1. Check the PT_GNU_STACK of the main binary ---------
int executable_stack = EXSTACK_DEFAULT;
for (i = 0; i < elf_ex->e_phnum; i++, elf_ppnt++)
  if (elf_ppnt->p_type == PT_GNU_STACK)     {
    if (elf_ppnt->p_flags & PF_X)
      executable_stack = EXSTACK_ENABLE_X;
    else
      executable_stack = EXSTACK_DISABLE_X; }
//---- 2. Set the memory permission -------------------------
#define TASK_EXEC \
    ((current->personality & READ_IMPLIES_EXEC) ? VM_EXEC : 0)
#define VM_STACK_FLAGS         (... | TASK_EXEC)  // expanded
unsigned long vm_flags = VM_STACK_FLAGS;
if (executable_stack == EXSTACK_ENABLE_X)
  vm_flags |= VM_EXEC;
else if (executable_stack == EXSTACK_DISABLE_X)
  vm_flags &= ~VM_EXEC;
mprotect_fixup(..., vm_flags);
```

through line 29 and lines 32-35. ③ If all object files have no `.note.GNU-stack` section, `notesec` will be always `NULL` and `elf_stack_flags` will not be updated at line 29. In that case, LD will not create any `PT_GNU_STACK` segment. ④ When only some object files have `.note.GNU-stack` sections but none of them has `SEC_CODE`, `notesec` will not be `NULL` and a `PT_GNU_STACK` segment will be created. Its execution permission depends on `exec`, which could be set to `PF_X` at line 26, based on `default_execstack` members of the backend data `bed` and the linker data `info`. Data `bed` is defined in an architecture-dependent macro `elf_backend_default_execstack`. In the latest version of LD, some architectures set it to `false`, like aarch64 and ia64, while others adopt the default value `true`, like x86, x64 and arm. Data `info` is configurable before we build the linker LD and its current default value is `true`.

We summarize the LD behavior on `PT_GNU_STACK` segment creation in Table IV. GOLD, a faster GNU linker, has a very similar behavior as LD. However, the LLVM linker LLD is different as its developers believe executable stack should be fully prohibited nowadays. If `.note.GNU-stack` does not exist, LLD will create this section without the executable flag. If `.note.GNU-stack` exists and has the executable flag, LLD will simply drop the flag. Using the command-line option `-z execstack` is the only way to enable executable stack with LLD. However, CLANG still uses LD as the default linker and LLD will only be used with an extra option `-fuse-ld=lld`.

All 11 affected IRM implementations work on x86, x64 or arm architectures. The assembler produces at least one object file without the `.note.GNU-stack` section, while all other object files have this section without the `SEC_CODE` flag. This situation falls into category ④ in Table IV, where LD will create the `PT_GNU_STACK` segment with `PF_X` in the final binary.

TABLE VI: Kernel logic for READ_IMPLIES_EXEC, based on the PT_GNU_STACK, the Linux kernel version and the CPU architecture.

PT_GNU_STACK	arm	Kernel	Kernel >= v5.8-rc1	
	< v6	< v5.8-rc1	x86/>= armv6	x64/aarch64
PF_R|PF_W	yes	no	no	no
PF_R|PF_W|PF_X	yes	yes	no	no
segment missing	yes	yes	yes	no

```
// Before Kernel v5.8-rc1 (all)
#define elf_read_implies_exec(ex, executable_stack) \
    (executable_stack != EXSTACK_DISABLE_X)

// After Kernel v5.8-rc1 (x86/x64)
#define elf_read_implies_exec(ex, executable_stack) \
    (mmap_is_ia32() && executable_stack == EXSTACK_DEFAULT)
// After Kernel v5.8-rc1 (arm)
int arm_elf_read_implies_exec(int executable_stack) {
  return (executable_stack == EXSTACK_DEFAULT ||
          cpu_architecture() < CPU_ARCH_ARMv6);     }
// After Kernel v5.8-rc1 (aarch64)
#define compat_elf_read_implies_exec(ex, stk) \
    (stk == EXSTACK_DEFAULT)
```

D. Main Binary Loading

When we create a new process on Linux, it takes two steps to load the main binary and dependent shared libraries into the process memory space. Both steps may change the permission of the stack and other regions based on different attributes.

In the first step, the Linux kernel will load the code and data sections of the main binary into memory. If the ELF binary specifies an interpreter in the PT_INTERP segment, like */lib64/ld-linux-x86-64.so.2*, the kernel will also load the interpreter into the memory. However, the kernel only checks the PT_GNU_STACK segment of the main binary to determine the executable permission of the stack. The interpreter's PT_GNU_STACK segment is ignored. Table V shows the kernel code for setting the stack permission for a new process. If the main binary contains the PT_GNU_STACK segment (line 4), the variable executable_stack will be updated accordingly at line 6 or line 8. When it comes to line 14 or line 16, the kernel will add or remove the VM_EXEC flag and invoke mprotect_fixup to set the page permission. However, when the main binary does not have a PT_GNU_STACK segment, executable_stack will have the default value, and the kernel will assign the VM_STACK_FLAGS, a complicated macro, to the stack. When we expand the macro VM_STACK_FLAGS, it will include VM_EXEC if the current process personality covers READ_IMPLIES_EXEC.

READ_IMPLIES_EXEC. BADASS will have the most significant impact on application security if it triggers READ_IMPLIES_EXEC, where all readable memory regions are set to executable, including but not limited to stack, heap, .bss and .data sections. This feature is designed to support special binaries compiled for old CPUs that lack NX bit, where any readable memory is assumed to be executable. Linux kernel determines to enable this devastating feature based on the availability of PT_GNU_STACK, the kernel version, and the CPU architecture. We analyze the related code in the Linux kernel, and summarize the results in Table VI. For any arm CPU older than ARMv6, the Linux kernel will always enable READ_IMPLIES_EXEC since these processors do not support NX bit. Besides, on Linux kernels before v5.8-rc1 (released on June 14, 2020), READ_IMPLIES_EXEC is enabled if the main executable does not have a PT_GNU_STACK segment or its PT_GNU_STACK segment has the PF_X flag. From kernel v5.8-rc1, only if PT_GNU_STACK is missing and the program runs on a 32-bit architecture, READ_IMPLIES_EXEC will be enabled. READ_IMPLIES_EXEC not only affects the binary and interpreter loading at the process creation, but also impacts all subsequent memory allocations through mprotect and mmap.

TABLE VII: Dynamic library loading results and related code.

Main binary	Any shared library (PT_GNU_STACK)		
PT_GNU_STACK	PF_R|PF_W	PF_R|PF_W|PF_X	segment missing
PF_R|PF_W	-	EXEC	EXEC (x86/x64/arm)
PF_R|PF_W|PF_X	-	-	-
segment missing	-	EXEC (aarch64)	-

```
1  //---- 1. Set dl_stack_flags based on main binary ----------
2  dl_stack_flags = DEFAULT_STACK_PERMS;
3                              // PF_R|PF_W      for aarch64
4                              // PF_R|PF_W|PF_X for x86/x64/arm
5  for (ElfW(Phdr) *ph = phdr; ph < &phdr[phnum]; ++ph)
6    if (ph->p_type == PT_GNU_STACK) {
7      dl_stack_flags = ph->p_flags;
8      break;                          }
9
10 //---- 2. Repeat for EVERY dynamically loaded library -------
11 unsigned int stack_flags = DEFAULT_STACK_PERMS;
12 for (ph = phdr; ph < &phdr[l->l_phnum]; ++ph)
13   if (ph->p_type == PT_GNU_STACK) {
14     stack_flags = ph->p_flags;
15     break;                          }
16 if ((stack_flags & ~dl_stack_flags) & PF_X)
17   __stack_prot |= PROT_READ|PROT_WRITE|PROT_EXEC;
18   if (mprotect (page, dl_pagesize, __stack_prot) == 0)
19     dl_stack_flags |= PF_X;
```

After merging the impact of READ_IMPLIES_EXEC, we summarize the executable permission of process memory regions in Table V. "exec-all" means all readable memory regions are executable due to READ_IMPLIES_EXEC, while "exec-stack" means that other than code sections, only the stack is executable due to the PF_X flag in PT_GNU_STACK. *"exec-all"* is the final result after merging "exec-all" and "exec-stack".

Eight out of 11 affected IRM implementations, including MCFI, RockJIT, πCFI and all five reassemblers, will set the PF_X flag of the PT_GNU_STACK segment in the main binary of hardened programs. When executing these hardened programs, once the main binary is loaded, the in-memory stack is set to be executable. The remaining three IRMs set PF_X in PT_GNU_STACK of shared libraries, and do not have an in-memory executable stack (yet) after loading the main binary.

E. Shared Library Loading

After loading the binary and the interpreter, the Linux kernel will transfer the control to the user-space interpreter, which recursively loads all shared libraries required by the main binary. For ELF binaries, the default interpreter is /lib64/ld-linux-x86-64.so.2, commonly called "loader". Table VII shows how the loader manipulates stack permission based on each shared library. It first sets a global flag dl_stack_flags to the architecture-dependent default value (line 2), which includes PF_X for x86, x64 and arm, but only contains PF_R and PF_W for aarch64. Then, the loader checks the PT_GNU_STACK segment of the main binary to update dl_stack_flags (line 7). After the initialization, the

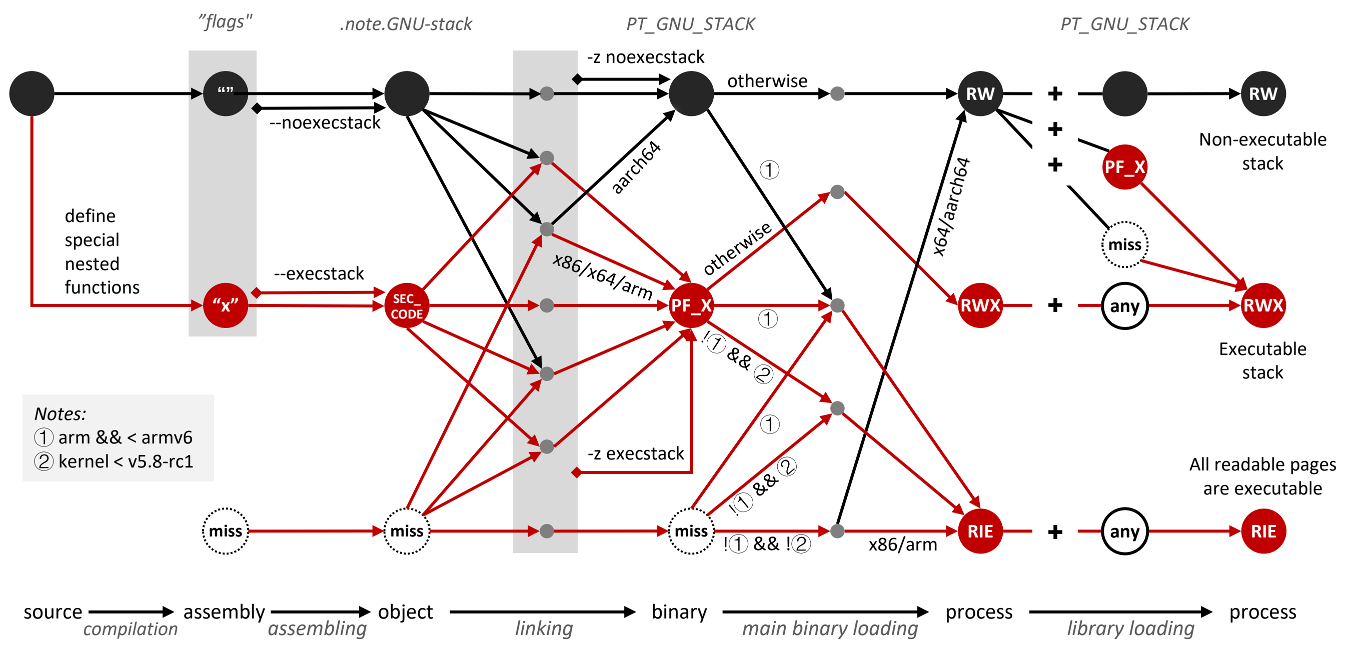


Fig. 5: Compilation details related to BADASS. The top workflow shows the normal compilation behavior that produces non-executable stacks. Red lines indicate possible root causes that introduce executable stacks or make all readable memory regions executable.

loader processes shared libraries one by one. For every library, the loader parses the flags in `PT_GNU_STACK` and sets the `stack_flags`. If `PT_GNU_STACK` is missing, `stack_flags` is set to the default value (line 11). If `stack_flags` has `PF_X` while `dl_stack_flags` does not have it, the loader believes an executable stack is required for the current shared library. It will set `__stack_prot` to `PROT_READ|PROT_WRITE|PROT_EXEC` (line 17) and invoke `mprotect` to make the stack executable. After the successful `mprotect`, the loader saves `PF_X` to `dl_stack_flags` and the code at line 16 will be `false`. In this way, the loader will not further alter the stack permission for subsequent libraries and the process stack will always be executable.

We summarize the behavior of the loader regarding executable stacks in Table VII, where "-" means no changes and "EXEC" indicates a `mprotect` syscall will make the stack executable. In general, the loader assumes the kernel has set up the stack permission based on the `PT_GNU_STACK` segment of the main binary (the purpose of `dl_stack_flags`). It examines all shared libraries and makes the stack executable as long as any library requires so. The `PT_GNU_STACK` segment of the loader is ignored by default. The only exception is that the user explicitly employs the loader to run the binary, like `/lib64/ld-linux-x86-64.so.2 ./hello_world`. In this case, the loader serves as the main binary and `hello_world` is treated as one library. Rules in Table V and Table VII will be applied.

In our investigation, PathArmor, ERIM and Donky are the three IRMs that set flag `PF_X` to segment `PT_GNU_STACK` in shared libraries instead of the main binary. All these shared libraries work as security runtime to protect the application. However, when these shared libraries are loaded, the loader will call `mprotect` to change the stack permission to executable.

F. Putting Everything Together

Based on the analysis of the compilation and binary-loading toolchain on Linux, we summarize the complicated logic related to stack executable permission in Figure 5. The first row shows the normal compilation workflow where no BADASS exists. Red lines demonstrate the potential locations where developers may miss necessary attributes and introduce executable stacks.

Although every component has a dedicated (and working) design to support W⊕X, each of them relies on the output of the previous tool to produce proper settings. We did not find any defect from the implementation. However, they form a long chain of trust and dependency to produce non-executable stacks. As long as one input does not follow the assumption, like one component fails to work or the file is not produced by the previous tool, the dependency chain will break and we may get a program with an executable stack. Assembly code is the most common reason to break W⊕X since developers occasionally write assembly files for various reasons such as security and performance. Due to the subtle design in assembling, it is easy for developers to miss the essential section and introduce executable stacks. We should notify developers about the unexpected security downgrades.

Takeaway: The W⊕X enforcement on Linux requires concerted and sophisticated collaborations of multiple compilation and execution tools, including the Linux kernel, which is complex and hard to follow. Hand-written assembly files join the compilation process in the middle, and break the regular W⊕X enforcement routine. The subtlety of the `.note.GNU-stack` section causes most developers to miss it in hand-written assembly files, which is the main root cause of executable stacks.

VI. Problem Generalization

The unexpected investigation result of BadAss in §IV motivates us to check other practical in-use security mechanisms similar to W⊕X, to understand whether they are also mistakenly disabled by experienced security developers. We select five widely adopted security mechanisms, specifically, PIE, RELRO, stack canary [32], FORTIFY_SOURCE [50] and Intel CET [69]. We adopt the same methodology as in §IV-B to check all IRMs in Table I. We get two new findings, where three CFI solutions mute FORTIFY_SOURCE and 11 IRMs disable Intel CET.

Disabling FORTIFY_SOURCE. FORTIFY_SOURCE provides a lightweight method to detect buffer overflow vulnerabilities related to common string copy or memory copy functions, like `memcpy` and `strcpy` [50]. With FORTIFY_SOURCE, compilers such as GCC will replace unsafe copy functions with corresponding safe wrappers, like replacing `strcpy` with `strcpy_chk`. The latter will check the buffer size during the compilation or at runtime, and terminate the execution once the destination buffer cannot store all data from the source. FORTIFY_SOURCE is enabled as long as the compilation configuration specifies any non-zero optimization level. Our investigation shows that three CFI solutions, specifically, MCFI/RockJIT/πCFI, will disable FORTIFY_SOURCE in the hardened program, leading to reduced capability to detect buffer overflow bugs. The root cause is that they wrap original indirect function calls to sensitive library functions with an inserted runtime function `__patch_call`. This replacement prevents compilers like GCC from detecting invocations of predefined memory-copy and string-copy functions, and therefore, compilers cannot replace them with secure wrappers. To address this issue, these CFI implementations can first replace insecure functions with secure wrappers, and then conduct CFI-related code transformations.

Suppressing Intel CET. Another finding is related to Intel CET [69], a hardware-assisted CFI solution proposed by Intel in 2017. Intel CET consists of two features, indirect branch tracking (IBT) for forward-edge CFI and shadow stack (SHSTK) for backward-edge CFI. Similar to W⊕X, enforcing Intel CET on Linux requires support from multiple parties, including the CPU, the compilation toolchain, and the Linux kernel. Intel activated the hardware support for Intel CET in the 11th generation Tiger Lake CPU, released in October 2020. IBT has been supported by the Linux kernel since v5.18 (released in May 2022) and enabled by default after v6.2 (released in February 2023). The support for SHSTK in kernel came a little bit late but finally got merged into v6.6 (released in October 2023). Similar to the enforcement of W⊕X, the GNU compilation toolchain requires a section-definition directive `.section .note.gnu.property,"a"` in the assembly file to generate an Intel CET-protected binary. With this directive, the assembler will create the `.note.gnu.property` section and set up two entries called `GNU_PROPERTY_X86_FEATURE_1_IBT` and `GNU_PROPERTY_X86_FEATURE_1_SHSTK`. The two entries indicate whether all executable sections are compatible with IBT and SHSTK. Missing these two entries will disable the protections of Intel CET on the program. We identify that all 11 IRMs suffering from BadAss also disable Intel CET since they miss the necessary `.note.gnu.property` section in assembly files. For PathArmor, ERIM, and Donky, the hand-written assembly files are compiled and linked into shared libraries. When loading these shared libraries to a CET-enabled process, the loader will check their CET marker and raise an error to stop the execution. For the other eight IRMs, the absence of the CET marker occurs in the main binary, which makes Intel CET silently disabled. Ddisasm patched this issue together with the BadAss problem in version 1.5.5.

During the investigation of Intel CET, we identify another two entries in `.note.gnu.property` related to program functionality and security. Since 11 IRMs miss this section, the hardened applications may have functionality or security issues. The first entry is `GNU_PROPERTY_STACK_SIZE`, which stores the maximum stack size allowed for the program. Developers can set the stack size via the linker option `-z,stacksize`, or assembly code. Missing this property may render the process exhaust the stack space and trigger stack overflow at runtime. The second entry `GNU_PROPERTY_NO_COPY` indicates to disable copy relocation on protected data symbols. Without this property, when the program code accesses an external variable defined in a shared library, the linker will copy the data from the library to a writable location of current binary. If the original data is read-only, copy relocation will make it writable, which may lead to severe attacks [45]. When a relocatable object sets this property, the linker will treat protected data symbols as defined locally and prevent copying the data to other executables. Developers can set this property via definitions in assembly files.

VII. Discussion

In this section, we first discuss the potential attacks towards introducing executable stacks. Next, we summarize existing efforts for mitigating BadAss and provide our suggestions to avoid it. Then, we briefly discuss the differences between our work and previous related posts. We also explain the W⊕X enforcement on Windows and macOS. Finally, we will discuss another recent security issue related to the `.note` section.

A. *Malicious Attacks via* BadAss

During the IRM investigation, we identify that all BadAss issues are introduced due to developers' benign mistakes — missing the `.note.GNU-stack` definition directive in assembly files. However, malicious attackers may abuse BadAss to deliberately introduce executable stacks to applications such as open-source software. Although an executable stack is not a complete attack, it could be a stealthy way to accomplish one critical part of exploiting prevalent memory-safety bugs [83]. Once attackers find any exploitable bug, they can easily build exploits through code injection instead of ROP or ret2libc.

Although this kind of attack may seem to be implausible, the recent notable `xz` backdoor has demonstrated the feasibility and practicality of these stealthy attacks [51]. In that event, the malicious attacker first gained the trust of the developers of the `xz`, and later became a maintainer. After that, he/she modified the compilation script file to inject malicious content into the application binary. The modification is minimal and stealthy, allowing the malicious behavior to be accepted into widely used applications and systems such as Fedora OS. In another backdoor instance, the attacker inserted an erroneous "." symbol into the compilation script to turn off Linux Landlock sandboxing [47]. Code reviewers overlooked this minor change and thus merged the commit into the online repository. These accidents show that malicious attackers like to take multiple minor steps to achieve advanced persistent threat (APT) attacks. We should take action before attackers abuse BADASS to compromise real-world systems.

The intricate enforcement of W$\oplus$X exposes multiple surfaces for attackers to introduce executable stacks. First, they can implement a desired feature through a special nested function or replace an existing function with a nested version. When the modified program is compiled by GCC, the process will have an executable stack. This method is more stealthy than directly uploading malicious code, since the nested functions have merely benign functionalities and thus can pass most manual code reviews and automatic scanning [2], [52]. Second, they have multiple methods to introduce an executable stack through assembly files. They can add a new assembly file that has an executable `.note.GNU-stack` or has no `.note.GNU-stack` directive, where the latter is likely preferred. If the source code already contains some assembly files, attackers can remove existing `.note.GNU-stack` directives or change the flags of `.note.GNU-stack` to `"x"`. To make this attack stealthy, attackers can either introduce minimal changes, like creating an empty assembly file by adding one command in Makefile, or hide the malicious modifications within a large chunk of code changes. Third, they may even upload object files or shared libraries with BADASS issues. Although directly uploading these files is uncommon and easily detected, attackers may take the stealthy method used in creating the `xz` backdoor. In particular, they can hide such object files or shared libraries within a binary-format test file, and then modify the compilation script to extract these files and merge them into the final binary.

Due to ethical considerations, we never test these new attack vectors against real-world code repositories. However, based on our observation and investigation of executable stacks in popular applications, we believe it is trivial for attackers to inject these BADASS-enabling files into open-source software.

B. Mitigations of BADASS

During our analysis of the W$\oplus$X enforcement on Linux, we notice that continuous efforts have been made to mitigate the BADASS issue. First of all, several researchers are also concerned about this issue and have spent significant efforts to mitigate it. For example, Kees Cook and Jamie Strandboge have conducted a broader investigation on programs with executable stacks and reported their findings to corresponding developers [82]. Second, developers of compilation tools and Linux kernel are providing more convenient options and implementing more comprehensive checks. ① In 2010, GOLD implemented an option `--warn-execstack` to remind the risk of executable stacks [31]. ② LD, the most widely used linker, adopted this design and set it as the default option in 2022 [24]. During our investigation, we find many developers notice this issue from the LD's warnings. Although they may not completely understand the issue, they fix the problem to avoid warnings. ③ More than that, LD supports extending this warning to all writable-and-executable segments via `--warn-rwx-segments`. ④ Most recently, in November 2023, LD added two new options, `--error-execstack` and `--error-rwx-segments`, for developers to convert any stack-permission warnings into errors [25]. ⑤ Within the kernel, `READ_IMPLIES_EXEC` is disabled after v5.8-rc1 even if segment `PT_GNU_STACK` has flag `PF_X` [29]. ⑥ When loading the main binary for a new process, the kernel will generate a warning if the process has an executable stack.

Unfortunately, we can still find many cases of executable stacks from even popular applications, even recently. To further mitigate this issue, we propose the following suggestions to application developers and software maintainers.

Application Developers. Developers have full control of the source code and compilation configurations. They can take various actions to enhance the application security. First, they can avoid writing nested functions that enable executable stacks, or rewrite existing ones to make stack non-executable. Second, when the application has to use assembly code, they can adopt inline assembly or always include a proper `.note.GNU-stack` directive in every assembly file. Third, developers can adopt specific compilation tools that produce more secure binaries, like CLANG which drops support to nested functions and LLD which generates non-executable stacks by default. Fourth, they should specify compilation options to explicitly disable executable stacks, like `--noexecstack` for AS and `-z noexecstack` for LD. At last, when compilation tools or operating systems produce warning messages regarding the application security, developers should promptly fix them.

Security Researchers. Security researchers should pay special attention to avoid BADASS issues. When implementing a new security solution, they should examine the application security after hardening, besides the performance and compatibility measurement. Once the tool is released to the public, normal users usually trust professional researchers and may adopt the tool to protect their applications. Therefore, we should try our best to avoid bringing new security risks to hardened programs.

Compilation Toolchain Developers. Developers of compilation toolchains can assist software developers in improving application security. First, they can change the default settings of security-related options to secure values. For example, based on our code analysis in §V, when particular directives are missing, many macros and variables related to stack permissions prefer executable stacks for compatibility purposes. However, with rapid program development, only a few legacy applications require an executable stack. CLANG even drops the support to nested functions. Although updating settings may introduce compatibility issues, it will immediately eliminate many security issues. We have seen such changes in the Linux kernel, where `READ_IMPLIES_EXEC` is disabled after v5.8-rc1 even if segment PT_GNU_STACK has flag PF_X [29]. Second, considering that nested functions are no longer actively used in modern systems, we suggest GNU toolchain developers evaluate whether we can cancel the backward support to nested functions. If the support is not necessary anymore, we can redesign W⊕X with a simpler method, *e.g.*, setting the stack to non-executable at the end of compilation, like via the execstack command. Another systematic mitigation could be allowing the assembler to further scan the hand-written assembly files. If neither the compiler nor the assembler finds any nested function in source code and hand-written assembly files, the assembler can set -noexecstack for every assembly file. In this way, the W⊕X enforcement will not rely on assembly directive while maintaining the compatibility.

Code Reviewers. Manual code reviewers and automatic security-analysis tools should take the risk of BADASS into consideration, and design specific rules to identify changes that may enable executable stacks. These behaviors include introducing special nested functions to source code, adding assembly files without proper .note.GNU-stack directives, changing compilation options, and committing object files and shared libraries. They can design test cases to verify the stack permission and reject suspicious uploads.

C. Related Discussions

BADASS is not completely unknown to the public. We search online and find some related posts about BADASS. Ian Lance Taylor [84] summarizes the W⊕X enforcement in the GNU compilation toolchain without detailed code analysis. Chris Wellons [92] focuses on explaining nested functions and briefly introduces the W⊕X enforcement. Alejandro Hernandez [46] discusses the READ_IMPLIES_ELF feature in detail. In this paper, we conduct a systematic W⊕X enforcement analysis by inspecting the source code of the compilation toolchain, the kernel, and the loader. Additionally, we investigate the root cause of recurring BADASS cases, provide mitigation suggestions, and generalize the BADASS issue.

D. W⊕X on Other Operating Systems

We investigate the W⊕X enforcement on Windows and macOS and find BADASS does not exist in these systems. For Windows, W⊕X (known as DEP) is enabled by default without any special assembly directive. Disabling DEP requires explicit assembly directives, where the compiler inserts .def __enable_execute_stack and call __enable_execute_stack to set part of the stack executable when compiling a program with nested functions. For macOS with Intel CPUs, executable stack is not related to any assembly directive and can only be enabled with a non-default linker option -allow_stack_execute. The linker will set a program header flag ALLOW_STACK_EXECUTION in the Mach-O executable, which is checked by the loader to set the stack permission. For macOS on Apple Silicon, the executable stack is completely disabled. This shows that Windows and macOS adopt more secure strategies to enforce W⊕X.

E. Security Issue within .note Section

In this paper, we identify two .note sections highly related to application security, *i.e.*, .note.GNU-stack and .note.gnu.property. It shows that developers are actively using structured ELF note sections to store critical program information. A recent security issue shows that storing sensitive information in the general .note section is also risky [30]. Kernel Address Space Layout Randomization (KASLR) is an effective address randomization technique for kernel space. It was first introduced to the Linux kernel in 2013. Kernel developers spent significant efforts to eliminate information leakage via kernel-user channels since attackers can leverage such information to bypass KASLR. Nowadays, inferring the kernel location is believed to be difficult. However, a security researcher recently noticed that kernel has stored the address of function startup_xen in its .note section. This section will be mapped to a user-space file called /sys/kernel/notes and the file is world-readable. Attackers just read this file to obtain a randomized address of startup_xen and then completely bypass KASLR. More importantly, this problem was introduced in 2007, long before KASLR was implemented. This demonstrates that the subtle design of the .note section may also lead to security downgrades. We should pay attention to these sections to avoid security breaches.

VIII. CONCLUSION

In this paper, we investigate program-hardening tools and inspect the source code of the compilation toolchain to understand the challenge of properly enforcing W⊕X in Linux systems. The result reveals that W⊕X on Linux relies on a long chain of trust and dependency to safeguard the process stack. It is challenging for developers, even security researchers, to notice the subtle design and may introduce executable stacks to popular and even hardened applications. To avoid similar accidental security downgrades in the future, we need to revise the design of W⊕X on Linux for simple and robust protection.

ACKNOWLEDGMENT

We thank the anonymous reviewers, especially our shepherd, for their insightful comments and valuable feedback. We also appreciate the assistance of the IRM developers for their detailed discussions and prompt resolution of reported issues. This research was supported by National Science Foundation (NSF) under grants CNS-2247652 and CNS-2339848. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005, pp. 340–353.

[2] P. Avgustinov, O. De Moor, M. P. Jones, and M. Schäfer, "QL: Object-Oriented Queries on Relational Data," in *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[3] "Consider Marking the Stack Non-Executable in Assembly Files," https://github.com/ROCm-Developer-Tools/clr/issues/22, Oct. 2023.
[4] "Linking with FatBinary Files Disables Stack Execution Protection," https://github.com/llvm/llvm-project/issues/71711, Nov. 2023.
[5] "Warning: Missing .note.GNU-stack Section Implies Executable Stack," https://github.com/cms-sw/cmssw/issues/44609, Apr. 2024.
[6] ".note.GNU-stack," https://news.ycombinator.com/item?id=11601725, Apr. 2016.
[7] "Hardened/GNU Stack Quickstart," https://wiki.gentoo.org/wiki/Hardened/GNU_stack_quickstart, May 2021.
[8] "Unbootable Grub2 on Gentoo with Lots of Compiler Warnings," https://savannah.gnu.org/bugs/?25220, Dec. 2008.
[9] "Avoid Executable Stack," https://github.com/ckolivas/lrzip/pull/243, Apr. 2023.
[10] "Mattermost-desktop Utilizes an Executable Stack," https://github.com/mattermost/desktop/issues/797, May 2018.
[11] "Insecure Writable and Executable Stack," https://github.com/mupen64plus/mupen64plus-core/issues/627, Feb. 2019.
[12] "Rocket.Chat Desktop App Uses an Executable Stack in Version 2.10.5," https://github.com/RocketChat/Rocket.Chat.Electron/issues/718, May 2018.
[13] "Build Results in Binary with Executable Stack," https://github.com/veracrypt/VeraCrypt/issues/146, Mar. 2017.
[14] "VSCode for Linux Utilizes an Executable Stack," https://github.com/microsoft/vscode/issues/49793, May 2018.
[15] "Wire Desktop App Uses an Executable Stack in Version 3.0.2816," https://github.com/wireapp/wire-desktop/issues/1464, May 2018.
[16] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2017, pp. 824–839.
[17] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy Sailr! There is No Need to Dream of C: A Compiler-aware Structuring Algorithm for Binary Decompilation," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, Philadelphia, PA, USA, Aug. 2024.
[18] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, "Superset Disassembly: Statically Rewriting x86 Binaries without Heuristics," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.
[19] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-reuse Attack," in *Proceedings of the 6th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, Hong Kong, China, Mar. 2011, pp. 30–40.
[20] P. Brady, "Libs Built with Executable Stack on Non AMD64 Architecture," https://github.com/facebook/zstd/issues/2963, Dec. 2021.
[21] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.
[22] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2015.
[23] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Oct. 2009, pp. 559–572.
[24] N. Clifton, "Emit a Note Warning the User that Creating an Executable Stack Because of Missing .note.GNU-stack Section is Deprecated," https://github.com/bminor/binutils-gdb/commit/0d38576a34ec64a1b4500c9277a8e9d0f07e6774, Apr. 2022.
[25] ——, "Add Ability to Change Linker Warning Messages into Errors when Reporting Executable Stacks And/Or Executable Segments," https://github.com/bminor/binutils-gdb/commit/e922d1eaa3774a68c96eae01e0fd08f8a30cda8c, Nov. 2023.
[26] K. Cook, "Forces Executable Stack When Used by Applications," https://bugs.launchpad.net/ubuntu/+source/link-grammar/+bug/409766, Aug. 2009.
[27] ——, "Mountall Has an Executable Stack," https://bugs.launchpad.net/ubuntu/+source/mountall/+bug/434813, Sep. 2009.
[28] ——, "non-exec stack markings," https://bugs.launchpad.net/ubuntu/+source/zip/+bug/375121, May 2009.
[29] K. Cook and B. Petkov, "X86/ELF: Disable Automatic READ_IMPLIES_EXEC on 64-bit," https://github.com/torvalds/linux/commit/9fccc5c0c99f238aa1b0460fccbdb30a887e7036, Apr. 2020.
[30] J. Corbet, "When ELF Notes Reveal Too Much," https://lwn.net/Articles/962782/, Feb. 2024.
[31] C. Coutant, "Add Warnings for Executable," https://github.com/bminor/binutils-gdb/commit/83e17bd5ed3c2586f558202172bf9f52ac80650c, Dec. 2010.
[32] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proceedings of the 7th USENIX Security Symposium (USENIX Security)*, San Antonio, TX, Jan. 1998, pp. 63–78.
[33] E. Dandrea, "Electron Has an Executable Stack," https://github.com/electron/electron/issues/11628, Jan. 2018.
[34] B. Darnell, "Security: Build with Non-Executable Stacks," https://github.com/cockroachdb/cockroach/issues/37885, May 2019.
[35] DARPA, "Cyber Grand Challenge (CGC)," https://www.darpa.mil/program/cyber-grand-challenge, 2016.
[36] L. Di Bartolomeo, H. Moghaddas, and M. Payer, "ARMore: Pushing Love Back into Binaries," in *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*, Anaheim, CA, USA, Aug. 2023, pp. 6311–6328.
[37] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, Virtually, May 2020, pp. 1497–1511.
[38] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary Rewriting without Control Flow Recovery," in *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Virtual, Jun. 2020, pp. 151–163.
[39] U. Erlingsson, *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Cornell University, 2004.
[40] U. Erlingsson and F. B. Schneider, "IRM Enforcement of Java Stack Inspection," in *Proceedings of the 21st IEEE Symposium on Security and Privacy (IEEE S&P)*, Oakland, CA, May 2000.
[41] D. Evans and A. Twyman, "Flexible Policy-directed Code Safety," in *Proceedings of the 20th IEEE Symposium on Security and Privacy (IEEE S&P)*, Oakland, CA, May 1999.
[42] A. Flores-Montoya and E. Schulte, "Datalog Disassembly," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, Virtually, Aug. 2020, pp. 1075–1092.
[43] GCC Online Manual, "Nested Functions," https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html.
[44] ——, "Support for Nested Functions," https://gcc.gnu.org/onlinedocs/gccint/Trampolines.html.
[45] X. Ge, M. Payer, and T. Jaeger, "An Evil Copy: How the Loader Betrays You," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.
[46] A. Hernandez, "A Short Tale About executable_stack in elf_read_implies_exec() in the Linux Kernel," https://ioactive.com/a-short-tale-about-executable_stack-in-elf_read_implies_exec-in-the-linux-kernel/, Nov. 2013.
[47] D. Hicks, "Xz: Can You Spot the Single Character That Disabled Linux Landlock?" https://news.ycombinator.com/item?id=39874404, Mar. 2024.
[48] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018, pp. 1470–1486.
[49] 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2016, pp. 969–986.
[50] J. Jelinek, "Object Size Checking to Prevent (Some) Buffer Overflows," https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html, Sep. 2004.
[51] "CVE-2024-3094 XZ Backdoor: All You Need to Know," https://jfrog.com/blog/xz-backdoor-attack-cve-2024-3094-all-you-need-to-know/#:~:text=TL%3BDR%20%E2%80%93%20the%20end%20goal,executed%20before%20the%20authentication%20step%2C, JFrog, Apr. 2024.
[52] W. Kang, B. Son, and K. Heo, "TRACER: Signature-Based Static Analysis for Detecting Recurring Vulnerabilities," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, USA, Nov. 2022, pp. 1695–1708.

[53] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014, pp. 147–163.

[54] K. Kylheku, "Warn HN: Stacks Made Executable on GNU by Mere Presence of Assembly Code," https://news.ycombinator.com/item?id=11599909, Apr. 2016.

[55] K. MacDermid, "Disable Executable Stacks on Assembly Objects," https://github.com/zerotier/ZeroTierOne/pull/2071, Aug. 2023.

[56] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015, pp. 941–951.

[57] J. Mason, S. Small, F. Monrose, and G. MacManus, "English Shellcode," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009, pp. 524–533.

[58] D. Maynor, "NX: How Well does it Say NO to Attackers' eXecution Attempts," https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-maynor.pdf, Las Vegas, NY, Jul. 2005, Black Hat USA Briefings.

[59] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *Proceedings of the 15th USENIX Security Symposium (USENIX Security)*, Vancouver, Canada, Jul. 2006, pp. 209–224.

[60] Microsoft, "Data Execution Prevention (DEP)," May 2023, https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[61] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: Better, Faster, Stronger SFI for the x86," in *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, Jun. 2012, pp. 395–404.

[62] J. R. Moser, "libffi Executable Stack (Missing .note.GNU-stack on .o Files)," https://gcc.gnu.org/bugzilla/show_bug.cgi?id=28036, Jun. 2006.

[63] Nergal, "The Advanced Return-into-lib(c) Exploits (PaX Case Study)," http://phrack.org/issues/58/4.html, Dec. 2001, Phrack.

[64] B. Niu and G. Tan, "Modular Control-Flow Integrity," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[65] ——, "RockJIT: Securing Just-In-Time Compilation using Modular Control-Flow Integrity," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2014, pp. 1317–1328.

[66] ——, "Per-Input Control-Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[67] A. One, "Smashing the Stack for Fun and Profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[68] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All You Ever Wanted to Know About x86/x64 Binary Disassembly but Were Afraid to Ask," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P)*, Virtually, May 2021, pp. 833–851.

[69] B. V. Patel, "A Technical Look at Intel's Control-flow Enforcement Technology," *Intel,[Online]. Available: https://www. intel. com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology. html*, 2020.

[70] D. Patel, A. Basu, and A. Mathuria, "Automatic Generation of Compact Printable Shellcodes for x86," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[71] PaX Team, "PaX Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[72] M. Prasad and T.-c. Chiueh, "A Binary Rewriting Defense against Stack based Buffer Overflow Attacks," in *Proceedings of the 2003 USENIX Annual Technical Conference (ATC)*, San Antonio, TX, Jun. 2003, pp. 211–224.

[73] S. Priyadarshan, H. Nguyen, R. Chouhan, and R. Sekar, "{SAFER}: Efficient and {Error-Tolerant} binary instrumentation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1451–1468.

[74] C. Qian, H. Hu, M. A. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A Framework for Post-deployment Software Debloating," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, USA, Aug. 2019.

[75] "Personality – Set the Process Execution Domain (READ_IMPLIES_-EXEC)," https://man7.org/linux/man-pages/man2/personality.2.html, Linux man-pages 6.04.

[76] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, Virtually, Aug. 2020, pp. 1677–1694.

[77] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2015.

[78] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, San Francisco, CA, Aug. 2011.

[79] 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 (CCS)*, Alexandria, VA, Oct.–Nov. 2007, pp. 552–561.

[80] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Jose, CA, May 2016.

[81] E. H. Spafford, "The Internet Worm Program: An Analysis," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 1, pp. 17–57, 1989.

[82] J. Strandboge, "ExecutableStacks," https://wiki.ubuntu.com/SecurityTeam/Roadmap/ExecutableStacks.

[83] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2013, pp. 48–62.

[84] I. L. Taylor, "Executable Stack," https://www.airs.com/blog/archives/518, Jun. 2011.

[85] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, San Diego, CA, Aug. 2014, pp. 941–955.

[86] "CockroachDB Customers," https://www.cockroachlabs.com/customers/, (visited in October 2023).

[87] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, USA, Aug. 2019, pp. 1221–1238.

[88] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015, pp. 927–940.

[89] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, Dec. 1993, pp. 203–216.

[90] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[91] S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2015, pp. 627–642.

[92] C. Wellons, "Infectious Executable Stacks," https://nullprogram.com/blog/2019/11/15/, Nov. 2019.

[93] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic Binary Recompilation," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Mar. 2020, pp. 133–147.

[94] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (IEEE S&P)*, Oakland, CA, May 2009.

[95] B. Zeng, G. Tan, and Ú. Erlingsson, "Strato: A Retargetable Framework for Low-level Inlined-reference Monitors," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2013, pp. 369–382.

[96] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical Control Flow Integrity and Randomization for Binary Executables,” in *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, May 2013.

[97] M. Zhang and R. Sekar, “Control Flow and Code Integrity for COTS Binaries: An Effective Defense against Real-world ROP Attacks,” in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, Dec. 2015, pp. 91–100.

[98] ——, “Control Flow Integrity for COTS Binaries,” in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, Aug. 2015.