Secure the Commodity Applications against Address Exposure Attacks

Xiaoguang Wang, Yong Qi
Department of Computer Science and Technology
Xi'an Jiaotong University, China
xiaogw@stu.xjtu.edu.cn, qiy@mail.xjtu.edu.cn

***Abstract*—Remote server vulnerability exploit is one of the most troublesome threat to the Internet security. An effective defense against the remote vulnerability exploit is code randomization, which randomizes the program code address to disrupt the malicious payload execution. Unfortunately, code randomization is particularly susceptible to address exposure vulnerabilities; the leak of a single code or data pointer is often sufficient to de-randomize the protected process. Existing solutions either prevent part of the address exposures (e.g., code-pointer exposure only), or are too heavyweight (e.g., have to involve a hypervisor software or a modified OS kernel).**

In this paper, we propose AXIS that can provide existing code randomization techniques with a comprehensive protection against address exposure. AXIS first redirects the code pointers through an indirection table that is protected by the execute-no-read memory segment. During the load time, all static data will be relocated to random locations, which breaks the fixed offsets between code and data. We have implemented a prototype of AXIS with only a customized compiler and a pre-loaded library. Our experiments show that AXIS can successfully eliminate address exposure with a minimal performance overhead.

1. Introduction

Remote server vulnerability exploit has been one of the thorniest problems in computer security researches. By feeding the vulnerable server with carefully constructed payloads, attackers can lead the victim process to execute the malicious code and finally take control of the server. Code randomization is an effective defense against attacks that rely on the exact locations or contents of the victim process [1]. The randomized code will not be loaded at a fixed address, which makes attackers difficult to construct the malicious payload. Indeed, a branch of researches have focused on randomizing the code at the granularity of code binary [2], functions [3], pages [4], basic blocks [5], etc. Address space layout randomization (ASLR), a coarse-grained form of code randomization which shifts each individual binary as a whole in the address space, has been widely adopted by major operating systems [6], [7], [8]. Because of its ubiquitous availability, ASLR has become one of our most important defenses in computer system.

Despite its effectiveness, code randomization (including ASLR) can be seriously weaken or even defeated by information exposure vulnerabilities [9], which leak the victim process' memory layout or contents. Address exposure is especially detrimental to ASLR systems. The leak of a single pointer (i.e., address) is probably enough to de-randomize ASLR, if the program binary is known. Most computers have a large part of their software come from a small number of vendors. For example, many Linux web servers are based on the LAMP stack (Linux, Apache, MySQL, and PHP/Perl/Python) from the leading Linux distributions. This homogeneity makes it relatively easy to figure out a remote software stack. In addition, a remote software configuration might be identified through fingerprinting, a technique that can distinguish software variations by their unique behaviors [10]. Accordingly, we assume a powerful yet realistic threat model in which the attacker can obtain the executable binaries of the target system. Program binaries contain a plethora of information useful for ASLR de-randomization. With the binaries, a leaked code or (static) data pointer allows the attacker to calculate the layout of the whole address space: code pointers include various types of instruction addresses, such as function pointers, return addresses, jump tables, and PLT/GOT tables. They all contain direct instruction addresses. Leaked data pointers are equally effective in disclosing the code layout because the code and the data have fixed offsets. In short, a complete solution to this problem should eliminate address exposure through both *code pointers* and *data pointers*.

Researchers have proposed solutions to solve code-pointer based address exposure and memory exposure. For example, XnR (eXecute-No-Read) system [11] modifies the Linux kernel to prevent (most of) the process code pages from being read. It leverages a sliding window to keep the latest n pages both readable and executable, while setting other code pages non-present. Readactor [12] further protects the code pointers from being leaked out by collecting the function pointers into execute-only pages, and uses a tiny hypervisor to make those memory pages execute-only. However, a global data pointer or pointers in PLT/GOT could also leak out the program memory layout. Fine-grained code randomization systems, such as [3], [4], [5], make attackers difficult to use a single leaked pointer to figure out the full process memory layout. However, the recent just-in-time ROP (JIT-ROP) attack [13] showed that

978-1-5386-1629-1/17/$31.00 ©2017 IEEE

the attacker can recursively reads the program code through a memory exposure vulnerability to discover all necessary gadgets on demand.

In this paper, we propose AXIS (address exposure elimination for ASLR), a system that can eliminate address exposure through both code pointers and data pointers. AXIS-enhanced ASLR systems are significantly harder to exploit than those not because address exposure will not leak critical information about the code layout. AXIS has two major components to turn a commodity server application into an *address exposure free* process. The first component is a customized compiler. It redirects the code pointers through an indirection table that is protected by the *execute-only memory segment*. This essentially substitutes the pointers (e.g., function pointers, function return addresses, exception tables, etc.) to the original program code with these to the indirection table. Therefore, a leaked code pointer accordingly will only reveal the indirection table addresses, but not the real code addresses. The second component is a standard loader extension. It relocates the static data to random locations. This breaks the fixed offsets between the code and the static data, making data pointer exposure much less useful to attackers. Static data consists of global data, PLT/GOT entries, as well as the indirection table generated by the first component. Dynamic data, such as the heap and the stack, have already been allocated at random locations by ASLR. Pointers to them have no direct relation to the code. We have implemented a prototype of AXIS based on the open-source LLVM compiler and a pre-loaded library. Our experiments with several popular benchmarks and applications show that AXIS can effectively reduce address exposure through both code pointers and data pointers. Our prototype only introduces a minor performance overhead (e.g., 3.3% on average for SPEC CPU2006). Overall, we consider the protection provided by AXIS is well worth this performance overhead for most applications.

The rest of this paper is organized as follows: Section 2 gives an overview of the address exposure problem. We then describe the design and the prototype of AXIS in Section 3. The evaluation is presented in Section 4. Afterwards, we discuss the related works in Section 5. Finally, we summarize the paper in Section 6.

2. Problem Overview

There are a wide spectrum of code randomization systems with different resilience to address exposure. In particular, ASLR only randomizes the bases of a program's executables. It does not change the internal layout or content of the code. ASLR is easy to implement and has the best compatibility – we only need to compile the program as a position-independent executable and the OS kernel takes care of the rest. However, it is also the most vulnerable to address exposure. If the attacker can de-randomize a function pointer (e.g., the `printf` function is located at address `0x400000`) or a pointer to some static data, he can infer the complete code layout because functions and data have fixed offsets from each other. Some other special pointers have

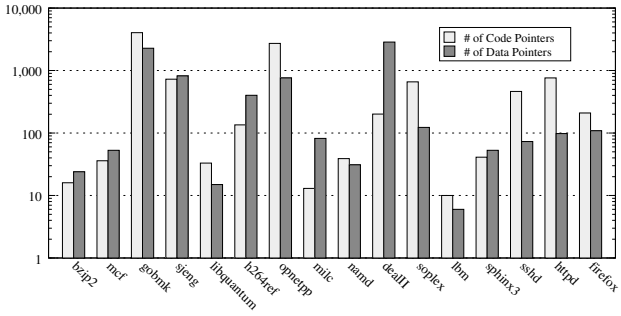

Figure 1: Code and Data Pointers in Several Applications

even worse effect on preserving address layout. For example, PLT/GOT entries contain addresses of external functions, which makes an attacker directly retrieve the code layout.

To quantify code and data pointers for a process, we write a simple program to dump the process' memory (via ptrace [14]) and scan it for code and data pointers. Specifically, we disassemble the program's code, and consider a memory word to be a code pointer if it points to the beginning of a valid instruction. We consider a memory word to be a data pointer if it points to the process' static data and is properly aligned (e.g., 4-byte aligned on 32-bit CPUs). Static data consist of global variables and static local variables, i.e., the data sections and the BSS sections. We do not count data pointers that point into dynamic sections, such as the heap and the stack, because ASLR randomizes their locations. Figure 1 shows the results of applying this tool to several benchmarks and real-world applications. More than half of the measured programs have hundreds or even thousands code and data pointers. Note that this is an overestimation of the actual numbers: the number of code pointers should be fairly precise because the x86 architecture has variable instruction lengths. The chance of a memory word collide with a specific instruction boundary is low. The number of data pointers is less precise, but should still be relatively close to the truth because most programs have small static data sections compared to their large address spaces (hence the low possibility of false positive). The results also depend on the program states, such as the current call stack. Overall, this figure demonstrates that address exposure through code pointers and data pointers is a realistic threat to ASLR.

Threat model: we assume a powerful attacker who has access to the executable binaries of a victim process. He can exploit address exposure vulnerabilities to obtain code and data pointers, and disambiguate them through data structure reverse-engineering. The target system may or may not support fine-grained code randomization, but ASLR is an integral defense (this represents the defense capabilities of popular commodity operating systems). Leaked pointers provide an effective pathway to de-randomize ASLR. However, we assume the target system could run on top of the secure kernel [15], [16], therefore, the underlying operating system can be trusted. Hardware side-channel attacks or other clever attacks to hijack the applications could emerge and require defences not described in this paper.

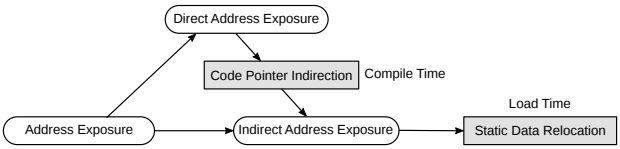

Figure 2: AXIS Overview

3. AXIS System

3.1. Overview

AXIS aims at improving ASLR by eliminating address exposure. An ASLR-protected process could be de-randomized through either direct address exposure or indirect address exposure. Code pointers are the major sources of direct address exposure because they all contain explicit instruction addresses. A leaked code pointer is catastrophic to ASLR in our realistic threat model (Section 2). To solve that, AXIS proposes a technique called code pointer indirection that transforms *direct address exposure* into *the indirect one* (Figure 2). Specifically, AXIS creates a proxy for each target of a code pointer to substitute it (common code pointer targets include function entry points and return sites). Code pointers now no longer carry addresses of the original targets, but these of the proxies. For example, we create a proxy for each function that has its address taken, and substitute the function address with the proxy address. A function pointer accordingly points to its proxy. All the proxies are collected in a dedicated section, called the indirection table. The indirection table has to be protected from memory exposure because its proxies carry the addresses of the original targets. Because each proxy is a short code snippet and no legitimate code needs to read it, we can protect the indirection table from *memory exposure* with the execute-only memory. However, the exposure of the proxy addresses could still de-randomize the process because each proxy lies at a fixed offset to the code. A leaked proxy address is just as effective as a leaked function entry point in divulging the code layout. This essentially turns the direct address exposure into the indirect address exposure, which is eliminated by AXIS' second technique – static data relocation. Even though code pointer indirection might seem to be a rather straightforward transformation, its interaction with PLT/GOT complicates the design (Section 3.2).

The second form of address exposure is caused indirectly by data pointer leaks. When the compiler builds a program, it lays out the code and the data at fixed relative locations. Offsets between program structures are hard-coded in the generated code. For ASLR to randomize a program, it must be compiled as a position-independent executable (PIE) (`gcc` and `llvm/clang` both use the `-fPIE -pie` flags for this purpose). A PIE program can be loaded at any location in the address space: instead of hard-coding code or data addresses, the program calculates their run-time addresses using the PC-relative addressing mode. For example, to access a global variable, the program first retrieves the current program counter (PC), and adds an offset to it to get the base of the Global Offset Table (GOT is a table generated by

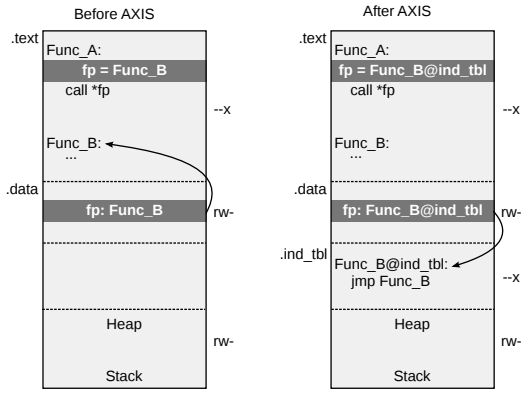

Figure 3: Function Pointer Indirection

the compiler to support dynamic linking [17]). The variable address is then calculated by adding a second offset to the GOT base [1]. When ASLR randomizes a program, it relocates the program as a whole and never changes the program's internal layout. Consequently, leaked data pointers (including pointers to the code indirection table) are equally harmful as leaked code pointers. To remove this indirect address exposure, we extend the dynamic loader to relocate *static* data sections (e.g., `.data`, `.bss`, and `.ind_tbl`) to random locations. This randomizes their fixed offsets to the code. Data pointers are no longer directly tied to the code.

By combining these two techniques, AXIS can eliminate address exposure through both code and data pointers. This significantly raises the bar to exploit ASLR-based systems. Our prototype of AXIS is source-code compatible with existing programs. There is no need to change the OS kernel or the system loader. Code pointer indirection is implemented as an extension to the LLVM compiler, while static data relocation is implemented as a shared library pre-loaded into the target process (using `LD_PRELOAD`). AXIS might also be implemented with a static binary re-writer to achieve the binary compatibility.

3.2. Code Pointer Indirection

Code pointers are the sources of direct address exposure. AXIS creates a proxy for each code pointer target, and converts code pointers to proxy pointers. All the proxies are aggregated in a dedicated section (`.ind_tbl`). Because proxies carry the addresses for the original targets in their code, we protect the `.ind_tbl` section from memory exposure with the execute-only memory. This will not cause compatibility issues because no legitimate code needs to read the proxies.

There are several types of code pointers, such as function pointers, return addresses, jump tables, and PLT/GOT entries. Jump tables are often used by compilers to speed up the `switch/case` statements. Each jump target entry handles one case of the statement. A function pointer and a jump

1. Even though the compiler could generate the variable address by adding its offset to the PC, it uses these two steps to reuse the GOT base. It is an expensive operation on x86-32 to get the current PC.

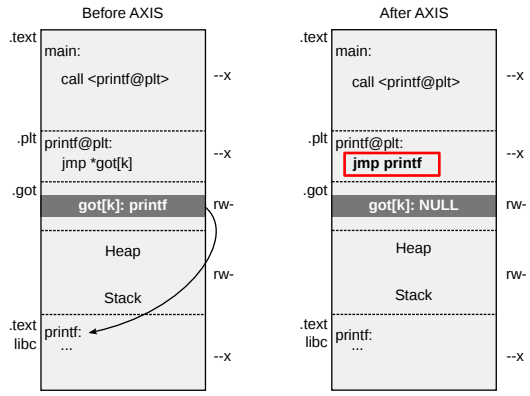

Figure 4: PLT Specialization for Direct Calls

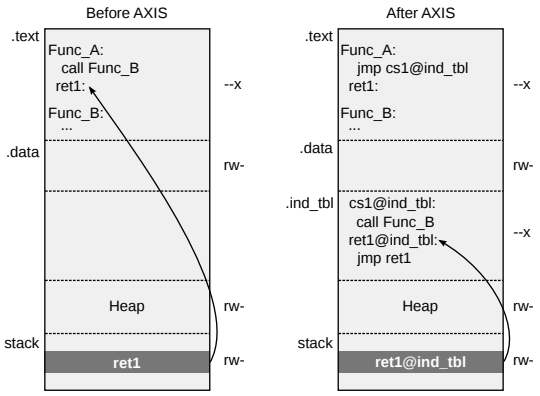

Figure 5: Return Address Indirection

table entry can be supported by AXIS in the mostly same way because they both carry a single instruction address (function address v.s. basic block address). AXIS compiler frontend converts function (and basic block) pointers to proxy addresses to prevent direct address exposure. A function proxy consists of simply a direct jump to the original function. Figure 3 shows how an indirect call through a global function pointer is instrumented by AXIS. In particular, the function pointer now carries the corresponding proxy address. At the load time, AXIS relocates the indirection table (the `.ind_tbl` section) to the random location, and further marks the indirection table execute-only. Therefore, attackers cannot retrieve the code locations with the function pointers. This process seems to be straightforward, but there are hidden pitfalls originated from the compiler's support of dynamic linking.

Today's programs rely heavily on common libraries for their functionality, performance, and convenience. Large programs like LibreOffice may utilize tens or even hundreds of libraries. A large amount of memory will be wasted if each program has its own copy of the used libraries. To address that, many systems allow libraries to be shared through dynamic linking: shared libraries are compiled as position-independent code that can be executed from any addresses, while programs are built with structures for dynamic linking, i.e., PLT/GOT (procedure linkage table/global offset table) [17]. These structures allow the program to link to the shared libraries that may have been loaded into the memory by another program (Figure 4 left). At the same time, they also introduce additional GOT-based direct address exposure: the GOT table contains function pointers that need to be dynamically resolved by the linker, and each PLT entry indirectly jumps to the function pointer in the associated GOT entry.

For direct calls to external functions, the compiler generates a PLT entry to represent a called external function. For example, a call to `printf` is substituted by a call to `printf@plt`, which indirectly jumps to the function address in the associated GOT entry (Figure 4). Notice that a PLT entry is essentially a proxy for the called function. We could repurpose PLT entries to eliminate GOT-based address exposure (Figure 4). Specifically, we configure the linker to aggressively resolve external function addresses during the program start and fill them into the GOT table. Our loader extension (Section 3.3) then replaces each applicable PLT entry with a *direct* jump the actual target functions, and wipes the related GOT entry. The PLT table is protected from memory exposure by the execute-only memory.

Another major source of direct address exposure are return addresses. Return addresses reside on the stack. Each return address points to a call site, i.e., the instruction after the call instruction. Return addresses are pushed to the stack by call instructions. We apply the same transformation to call sites as to functions: a proxy is created for each call site and the return address is replaced by the proxy address. Figure 5 shows how return address indirection works. A proxy for call site consists of two instructions: a call instruction to the original function and a direct jump back to the call site. The original call instruction is changed to a direct jump to the proxy. At the run-time, when the function is called, the program jumps to the proxy, which pushes the return address to the stack and transfers to the called function. The return address thus points to the proxy's jump instruction, instead of the real call site. When the function returns, the processor pops the return address off the stack and executes the corresponding jump instruction, which returns back to the original call site. Therefore, any pointers on the stack will point to a indirection table, instead of a code location. However, proxy addresses (a.k.a. indirection table) can still leak the code layout because they have fixed offsets to the code. Our next technique, static data relocation, prevents these indirection address exposure.

3.3. Static Data Relocation

Code addresses might be exposed indirectly through pointers to static data (i.e., data not on the heap or the stack). The compiler hard-codes the offsets between the code and the data. ASLR does not change these offsets. It instead moves the program binaries as a whole. Consequently, data pointer leaks are as harmful as code pointer leaks: the attacker only needs to add the correct offset to a leaked data pointer to locate a gadget. To prevent indirect address

```
0617: e8 00 00 00 00     call  61c<main+0xc>
061c: 58                 pop   %eax
061d: 81 c0 e4 19 00 00  add   $0x19e4,%eax
```

Figure 6: Computing GOT Base

exposure, AXIS leverages the unique structure of ASLR-compatible programs to randomize these offsets.

Programs must be compiled as position independent executables (PIE) to benefit from ASLR. Like shared libraries, PIE can be executed from any addresses. This is enabled by two key PIE structures. First, the dynamic linker resolves the run-time symbol addresses and stores them in the PLT/GOT tables; Second, the program uses PC-relative (program counter) addressing mode to reference data and functions. For example, if the program wants to assign a local function to a function pointer, it adds an offset to the current PC to get the function address and stores that in the pointer. To be precise, the function or data address is calculated relative to the GOT table base, which in turn is calculated relative to the current PC. This allows the program to reuse the GOT base.

Unfortunately, the 32-bit x86 architecture has no direct PC-relative addressing instructions. The PC-relative addressing mode is simulated by the compiler's built-in functions using the `call` instruction. For example, Figure 6 shows a popular method to compute the GOT base. Specifically, it makes a direct call to *the next instruction*. That is, the address of the next instruction (`pop %eax`) is pushed to the stack by the `call` instruction and further poped into register `eax`. `Eax` thus contains the address `0x61c`. An offset is then added to the current program counter to get the GOT base. The x86-64 architecture directly supports the PC-relative address mode. The structure in Figure 6 can be replaced by a single instruction, such as `lea 0x1ba7(%rip),%rsi`. Now, the program can access its data and functions independent of its load position.

AXIS relocates static data sections to prevent indirect address exposure. These sections include `.data`, `.bss`, `.got`, `.plt`, `.ind_tbl` (the indirection table of Section 3.2) etc. Dynamic sections like the heap and the stack do not need to be relocated because ASLR randomizes their locations during allocation. To relocate sections, AXIS first configures the linker to resolve all the symbols, and then relocates them using a pre-loaded library. Pre-loaded libraries execute before the main function. Accordingly, there is no need for AXIS to modify the system linker/loader. AXIS also collects information from the program binary (e.g., relocations) and the `/proc` directory (e.g., the memory layout). For each static data section, AXIS allocates an equal sized block of memory at a random location with the `mmap` system call. AXIS then copies the memory from the old section to this new location, and fixes the affected instructions that access these data (e.g., the instruction pattern shown in Figure 6). To guarantee that no old data will ever be accessed again, AXIS unmaps the original data sections from the process' address space.

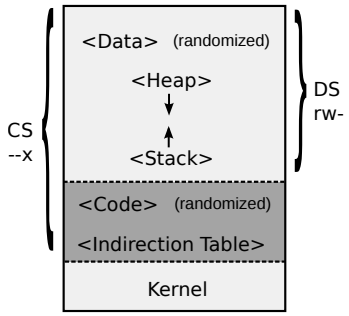

Figure 7: Segments Reorganized for Execute-only Code Memory

3.4. Execute-only Memory

Execute-only memory plays an important role in defending ASLR against information exposure vulnerabilities. For example, it can prevent JIT-ROP from reading the code of a finely randomized program. However, current x86 processor does not explicitly support the execute-only memory page, since executable code pages are always readable. Researchers have explored to use EPT in hypervisor layers to enable execute-only memory for a regular process [12]. However, the introduced hypervisor layer makes that solution bloated. AXIS makes a novel use of the traditional x86 segmentation, by reorganizing the `CS`/`DS` layout to enable the execute-only memory.

Segmentation has been proposed long ago to support the early x86 virtual memory system, but it was later superseded by paging [18]. For backward compatibility reasons, current operating systems set each of the segment's boundary as the whole virtual address space. For example, the code and data segments (confined by the segment selectors pointed by `CS` and `DS` registers respectively) in 32-bit processor mode are all set with address range of 0∼4GB. In AXIS, the loader extension re-configures the `CS` and `DS` segments, making data segment smaller than code segment. At the same time, the code segment is marked execute-only. The memory ranges that belong to the code segment while stay out of the data segment are now execute-only (dark black area in Figure 7) [2].

Within the execute-no-read memory segment, the program code can access the data or be executed, but the code itself cannot be read. All the above mentioned can be done by the AXIS loader extension to prepare and load the customized segment descriptors in the local descriptor table (LDT). The loader extension also configures the code segment to mark it execute-only (this can be configured in the `type` filed of the code segment descriptor [18]). After that, any attempts to read the execute-only memory would cause *a segmentation fault* and can be easily captured by AXIS.

2. Note that we cannot simply mark `CS` as execute-only without shrinking `DS`, because the process by default will use `DS` to make a data read.

3.5. Prototype of AXIS

We have built a prototype of AXIS based on the open-source, modular LLVM compiler infrastructure. Many programming languages, such as C/C++, D, Go, and FORTRAN, have a LLVM front-end that generate LLVM IR (intermediate representation). LLVM IR is a flexible form for code analysis and transformation. We use `Clang`, a C language family front-end for LLVM, as our front-end. Function pointer indirection is implemented in both the front-end and the back-end: the front-end analyze the code to collect function pointer targets, such as internal functions that have their addresses taken and external functions, while the back-end performs the actual transformation (in `AsmPrinter`). Return address indirection is implemented solely in the back-end, also in `AsmPrinter`. We assign each proxy to the `.ind_tbl.text` section. This instructs the compiler to put all the proxies in the same section to facilitate its relocation. Meanwhile, we implement static data relocation and execute-only memory in a shared library pre-loaded into the process' address space with `LD_PRELOAD`. A pre-loaded library executes before the main function. It thus can safely relocate the static data and prepare the segmentation. The library unloads itself afterward to prevent itself from being targeted by code reuse attacks.

4. Evaluation

In this section, we first analyze the security improvements to ASLR made by AXIS, and then evaluate its performance with standard benchmarks.

Security Evaluation: AXIS prevents the code pointer leak by redirecting the code pointers to the execute-only memory. In particular, code pointer targets, such as function entry points and return sites, are substituted by proxies. Thus the attackers can no longer read the real code locations from the pointers. Proxies and the static data will be relocated to the random memory locations by AXIS loader extension. Therefore, an attacker cannot use the fixed offset from static data and the code to deduce the code locations.

We empirically measure the effectiveness of code pointer indirection with the tool mentioned in Section 2. Specifically, we obtain the memory of a AXIS-protected program and count how many code pointers point to the code section (unprotected) and how many point to the proxies (protected). We found that the vast majority of code pointers have been protected. Nevertheless, there are a small number of leftovers. For example, we protected 755 code pointers out of all the 762 ones for the Apache web server. The other 7 code pointers are in fact all introduced by the compiler under the hood: 1) the PC-relative addressing mode in Figure 6 temporarily saves the current program counter to the stack. 2) the compiler inserts a number of functions to bootstrap the process (e.g., `__libc_start_main`). These functions are linked to every program as binaries, and thus are not converted by AXIS. To address that, we could provide our own instrumented bootstrapping functions, and ask the compiler to link to them instead of the built-in ones. A quick-and-dirty solution is to overwrite these functions upon the entry to the `main` function since they are no longer needed afterwards.

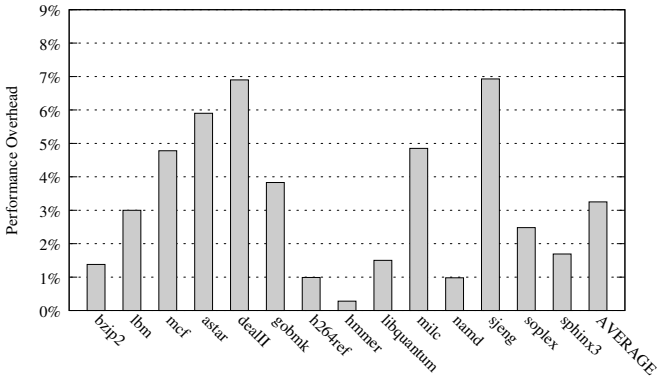

Figure 8: Performance Overhead of SPEC CPU2006

Performance Evaluation: We evaluate the performance of our prototype with standard SPEC CPU2006 benchmark. SPEC CPU2006 is a set of CPU-intensive applications to measure the system performance. AXIS' impacts to the performance are twofold: first, AXIS introduces an additional layer of indirection for code pointers, slowing down the programs. Second, AXIS changes the program's memory layout. Specifically, it aggregates proxies in a dedicated section and moves static data sections to random locations. Memory layout can affect the program's caching behaviors and the performance. Figure 8 shows the performance overhead of SPEC CPU2006 caused by AXIS. Sjeng has the largest overhead at 6.9%, while hmmer has the lowest overhead at 0.3%. On average, AXIS incurs about 3.3% performance loss for SPEC CPU2006. Overall, we consider the performance of AXIS is more than acceptable for most applications and the protection provided by AXIS is well worth this small performance loss.

5. Related Work

In this section, we summarize the recent researches to prevent the memory exposure attacks. Code randomization has been adopted by commodity operating systems in the form of coarse-grained address space layout randomization (ASLR), where program binaries are loaded at random bases [6], [7], [8]. However, ASLR has limited randomness on the 32-bit architectures [19], as well as vulnerable to information exposure attacks. To address that, fine-grained code randomization systems have been proposed to mingle the code at the page, function, or basic block level [3], [4], [5]. For example, Oxymoron proposes a new calling convention in which code pages are position and layout agnostic [4]. This allows Oxymoron to finely randomize code pages without losing the memory saving of shared libraries and executables. However, a single code-pointer based address exposure could circumvent this fine-grained protection by JIT-ROP [13], in which a single leaked pointer

can be used to recursively read out the whole process memory.

To address that, researchers proposed to implement execute-only memory pages in both software [11], [20], [21] and hardware [12], [18], [22]. For example, XnR controls the page table of the protected process so that only the currently active code pages are readable [11]. Readactor realizes the execute-only page with the help of the hypervisor's second-level address translation (a.k.a. Intel EPT) [12]. It further protects the code pointers from leaking the memory layout. However, besides the bloated hypervisor software, Readactor cannot prevent the data pointers from leaking out the memory layout. As demonstrated in Figure 1, there are plenty of data pointers in these processes. Static data always have fixed offsets to the program code, which could also leak out the process layout. AXIS solves both code and data pointers leakage problems by using a pointer indirection table as well as a static data relocation library. In addition, AXIS can more securely support dynamically linked programs, which will disclose the locations of *unredacted* libraries.

6. Summary

We have presented the design, implementation, and evaluation of AXIS, a practical system to improve ASLR's defense against address exposure attacks. ASLR is one of our most important defense against attacks that has been widely adopted by commodity operating systems. AXIS can prevent both direct address exposure via code pointers and indirect address exposure via data pointers. It has two key techniques, code pointer indirection and static data relocation. Code pointer indirection introduces a layer of indirection in order to stow real instruction addresses in the execution-only memory, while static data relocation randomize the offsets between the code and the data so that data pointers cannot be used to calculate the code location. Our evaluation demonstrates that AXIS can eliminate (most of) both code and data pointers and significantly raise the bar for working exploits. The performance evaluation with standard benchmarks and applications shows that our prototype incurs very minor performance overhead.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work was partly supported by the National Science Foundation of China (No. 61672421, 61402358, 61602363), the Ph.D. Programs Foundation of Ministry of Education of China (No. 20120201110010) and the China Postdoctoral Science Foundation (No. 2016M590927).

References

[1] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[2] PaX Team. PaX Address Space Layout Randomization (ASLR), 2003.

[3] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-grained Randomization of Commodity Software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[4] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. *Proc. 23rd Usenix Security Sym*, pages 433–447, 2014.

[5] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.

[6] Apple. OS X MountainLion Core Technologies Overviewe. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf.

[7] Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/.

[8] Alex Ionescu Mark Russinovich, David Solomon. *Windows Internals, 6th Edition*. Microsoft Press, 2012.

[9] CWE. CWE-200: Information Exposure. http://cwe.mitre.org/data/definitions/200.html.

[10] Nmap. Remote OS Detection. http://nmap.org/book/osdetect.html.

[11] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[12] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[13] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[14] Wikipedia. Ptrace. http://en.wikipedia.org/wiki/Ptrace.

[15] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.

[16] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. SecPod: A Framework for Virtualization-based Security Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, 2015.

[17] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, 1999.

[18] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual*, Feb 2014.

[19] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, 2004.

[20] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. 2015.

[21] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. Exoshim: Preventing memory disclosure using execute-only kernel code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*, pages 56–66, 2016.

[22] ARM: the Architecture for the Digital World. http://www.arm.com/.