The Leakage-Resilience Dilemma

Bryan C. Ward[1], Richard Skowyra[1], Chad Spensky[2], Jason Martin[1], and Hamed Okhravi[1](✉)

[1] MIT Lincoln Laboratory, Lexington, USA
{bryan.ward,richard.skowyra,jnmartin,hamed.okhravi}@ll.mit.edu
[2] University of California, Santa Barbara, USA
cspensky@cs.ucsb.edu

Abstract. Many control-flow-hijacking attacks rely on information leakage to disclose the location of gadgets. To address this, several *leakage-resilient* defenses, have been proposed that fundamentally limit the power of information leakage. Examples of such defenses include address-space re-randomization, destructive code reads, and execute-only code memory. Underlying all of these defenses is some form of code randomization. In this paper, we illustrate that randomization at the granularity of a page or coarser is not secure, and can be exploited by generalizing the idea of partial pointer overwrites, which we call the Relative ROP (RelROP) attack. We then analyzed more that 1,300 common binaries and found that 94% of them contained sufficient gadgets for an attacker to spawn a shell. To demonstrate this concretely, we built a proof-of-concept exploit against PHP 7.0.0. Furthermore, randomization at a granularity finer than a memory page faces practicality challenges when applied to shared libraries. Our findings highlight the dilemma that faces randomization techniques: course-grained techniques are efficient but insecure and fine-grained techniques are secure but impractical.

1 Introduction

Memory-corruption attacks continue to be one of the primary attack vectors against modern computer systems [2]. The sophistication of memory-corruption attacks has increased from simple code injection [38] to various forms of code-reuse attacks [11,43] in response to widespread deployment of defenses such as $W \oplus X$ (a.k.a. Data Execution Prevention – DEP).

Leakage-resilient memory-protection techniques [4,7,12,14,35,50,53] are considered the state-of-the-art in one of several approaches to mitigate the

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.
This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

© Springer Nature Switzerland AG 2019
K. Sako et al. (Eds.): ESORICS 2019, LNCS 11735, pp. 87–106, 2019.
https://doi.org/10.1007/978-3-030-29959-0_5

impact of memory corruption attacks. Such techniques protect the code against various forms of information-leakage attacks (*i.e.,* direct [45,47], indirect [16,41], or side-channel-based [8,42]), thus ensuring that the effects of the underlying randomization cannot be sidestepped by an attacker. Leakage-resilient techniques include various forms of execute-only techniques via memory permissions [4,14] or destructive reads [50], code-pointer protection via code and data decoupling [35], and runtime re-randomization techniques [7,12,53].

All of these leakage-resilient techniques crucially rely on the underlying code-randomization mechanism and its granularity. For example, execute-only memory can be easily bypassed if an attacker knows the code-section layout. Code-randomization techniques fall into two categories: virtual-memory randomization and physical-memory randomization. Virtual-memory randomization only changes the mapping of virtual addresses to physical addresses, and does not change the contents of physical memory. Because such mapping can only be as fine as a page, virtual-memory-randomization mechanisms have page-level granularity or coarser. Examples of such mechanisms include library-level randomization [7,39] and page-level randomization [5]. The second category, physical-memory randomization, is any technique that changes the contents of physical memory. These include function-level [23,31], basic-block-level [12,51], and instruction-level [18,30] randomization mechanisms.

In this paper, we study the security and practicality tradeoffs of code randomization for leakage-resilient defenses. We first show that virtual-memory randomization provides insufficient security guarantees. Extending the idea of partial pointer overwrites, we illustrate an attack, which we call *Relative ROP (RelROP)*, that can bypass such techniques in the absence of additional, protection mechanisms. Specifically, we show that by simply overwriting the least-significant bytes of a pointer, an attacker can address sufficient gadgets within a page to build an exploit, and because the granularity of virtual-memory randomization cannot be finer than a page, this limits their effectiveness in practice.

Although the idea of partial pointer overwrites existed in the literature before [8,19], building a complete attack based on them faces a number of challenges, including the difficulty of chaining gadgets together and the lack of access to many gadgets due to randomization of their addresses. To overcome these challenges, we illustrate how the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) can be abused as a layer of indirection to facilitate exploitation. We show that function pointers within the GOT may be partially overwritten to point instead to gadgets within the page of the original target. We illustrate that numerous gadgets are accessible in each page through partially overwriting GOT entries. We analyze many popular Linux applications and find that many such gadgets can be invoked while the system is protected by code randomization and many different leakage-resilient defenses.

To further demonstrate the realism of RelROP, we build a proof-of-concept exploit against PHP (Sect. 6), which deterministically bypasses many leakage-resilient defenses that rely on virtual-memory randomization.

We then investigate physical-memory randomization mechanisms. While these techniques can be arbitrarily fine-grained, and are thus secure against partial-overwrite attacks, they face many practicality challenges. Among them, is the fact that such techniques require actually moving memory contents, which creates challenges for shared libraries. Such challenges give rise to tradeoffs between security and performance or practicality.

Our findings highlight the dilemma when designing leakage-resilient memory-protection techniques, and illustrate that design choices must consider a fine trade-off between security and practicality in this domain. Since all of the proposed techniques in this domain face either security challenges or practicality challenges (or both), we posit that more research is needed to build effective and efficient leakage-resilient techniques.

The contributions of this paper are as follows:

- We provide an in-depth study of security and practicality implications of code randomization in leakage-resilient memory-protection techniques.
- We illustrate that virtual memory-based code randomization provides insufficient security. We leverage the idea of partial pointer overwrite to build a generic attack, called RelROP, that overwrites one or two least significant byte of a code pointer to access gadgets within the same page as the target.
- We conduct extensive analysis of the prevalence of RelROP gadgets, and find that sufficient RelROP gadgets are found in 94% of analyzed binaries.
- We show the realism of RelROP via a proof-of-concept exploit against PHP.
- We discuss the practicality challenges of physical-memory-based randomization techniques and argue that security and practicality trade-offs need to be considered when leveraging code randomization for leakage resilience.

2 Randomization Granularity

Leakage-resilient techniques, including TASR [7], Shuffler [53], Remix [12], Isomeron [16], Oxymoron [5], Heisenbyte [50], NEAR [52], Morton *et al.* [36], XnR [4], and HideM [22] mitigate the impact of information-leakage attacks on code randomization/diversification. They employ various mechanisms including memory permissions [4,14], destructive reads [50], code pointer protection [35], and runtime re-randomization [7,12,53] to prevent direct memory disclosures (*e.g.,* [4,5,22,52]) and sometimes both direct and indirect memory disclosures (*e.g.,* [7,12,14,16,35,53]).

A key component of every leakage-resilient scheme is a one-time randomization of memory (or more, in the case of re-randomization) in order to obscure the memory layout from the attacker. Once obscured, the remainder of the technique (*e.g.,* execute-only memory) seeks to ensure that the attacker cannot leak memory in order to discover the memory layout.

2.1 Virtual-Memory Randomization

One approach to randomizing memory is to randomize the mapping between virtual- and physical-memory addresses. Attackers relying on code reuse must

know the virtual-memory address at which physical code pages are mapped. This is the driving principle behind ASLR [39] and its descendants, for example.

Randomizing virtual addresses is straightforward, as only the page tables for that process need to be changed rather than the underlying physical memory (*i.e.,* no memory moves or copies are required). Therefore, such randomization can be performed efficiently, and ensures that physical pages mapped into multiple processes (*e.g.,* shared libraries) experience no disruption.

Randomization Granularity. Relying on virtual-memory randomization imposes a fundamental limitation on the granularity of randomization. Objects smaller than a page of memory cannot be independently randomized, as page tables cannot be used to reference the addresses of memory objects smaller than a page. Thus, some of the low-order bits of an address remain unchanged after randomization. While the exact size of memory pages is architecture-specific, 4KB is the smallest page size supported by common architectures such as x86, x86-64, and ARM.

In practice, defenses using virtual-memory randomization operate on the library- or page-level. Library-level is the most coarse-grained approach to memory randomization, in which the application binary and base addresses of shared libraries are randomized. It is implemented at load-time by ASLR [39]. TASR [7] provides a leakage-resilient version by re-randomizing in response to input/output system-call pairs. Note that in either case, all memory objects within a library remain at fixed relative offsets to one another, but the relative offsets among libraries are randomized.

Page-level randomization, implemented by Oxymoron [5] at load-time, attempts to provide enhanced security by randomizing at a finer granularity. This ensures that inter-page offsets are randomized, but leaves intra-page offsets fixed.

2.2 Physical-Memory Randomization

Rather than change virtual-to-physical mappings, a randomization technique can instead reorder data/code in physical memory. This requires memory copies that induce overhead, but can operate at an arbitrary level of granularity. Physical memory randomization must also account for how randomization of shared pages is handled, since different processes may be simultaneously attempting to access them. This can have both security and practicality implications.

Randomization Granularity. Unlike virtual-memory randomization, physical-memory randomization may operate at any level of granularity.[1] This can dramatically limit, or entirely remove, the availability of gadgets near code pointers. Recall that low-order bits are fixed in virtual-memory randomization, because

[1] In practice, physical-memory randomization has only been applied at the sub-page level, as virtual-memory randomization is more efficient for coarser granularities.

addresses are necessarily page-aligned (*i.e.,* the lower 12 bits are an offset into a page, and the upper bits specify the page in a 4K-size page).

Physical memory randomization does not have this constraint (as it does not rely on page tables), and can fully randomize the address of a memory object. For example, it could shift a function by a single byte. This would modify every bit in the address of that function, preventing an attacker from using their local copy of an application to infer anything about the victim's memory layout.

Physical-memory-randomization defenses have been presented at the function [53], basic-block [12,51], and instruction [26] randomization levels. Shuffler [53] randomizes the base address of all functions in a process image. Shared libraries are statically linked at load-time, in order to ensure that their functions can be safely relocated. Remix [12] and Binary Stirring [51] both randomize at the basic-block level. The former re-randomizes periodically, while that latter performs a single load-time randomization. ILR [26] uses process-level virtualization to randomize at the instruction granularity on program load. None of these approaches randomize shared libraries. We will discuss why later in Sect. 8.

3 Threat Model

We assume that a remote attacker has access to a memory-corruption vulnerability that enables arbitrary read and write access to userspace memory. This is consistent with common vulnerabilities that, for example, give attackers control over a buffer index (*e.g.,* CVE-2016-0034), or do not properly safeguard format strings (*e.g.,* CVE-2015-8617).

We make the following assumptions about the defensive configuration of the victim process. (1) W$\oplus$X is deployed on the system being attacked, so that code injection and code modification are prevented. (2) A leakage-resilient defense is deployed that prevents direct memory disclosures (*i.e.,* leakage of code pages). (3) The Global Offset Table exists. A GOT exists as long as shared libraries are used, and is even present for an isolated binary if it is compiled to be position-independent. Additionally, the majority of leakage-resilient defenses identified in this paper do not extend protections to the GOT, with the exceptions of Oxymoron [5] and Readactor [14]. In Sect. 8, we discuss the implications of requiring GOT protection in more detail. (4) The layout of code regions in memory have been randomized, so that the attacker does not have *a priori* knowledge of the location of code in memory.

This threat model is consistent with that of existing leakage-resilient defenses.

4 Relative ROP Attacks

In this section, we describe a code-reuse attack that generically circumvents many leakage-resilient defenses that rely on virtual-memory randomization. We show that an attacker can use existing code pointers to launch meaningful exploits. This is achieved by partially overwriting the low-order byte of code pointers such that they point to a relative offset within the randomized region,

without knowing or needing to corrupt the randomized high-order bytes of that pointer. Thus, we refer to these attacks as *Relative ROP (RelROP).*

4.1 Partial Pointer Overwriting

A critical assumption to the security of memory randomization is that pointers can only be corrupted *in toto*. However, pointers in modern architectures are not atomic, and in fact require multiple bytes of memory to encode. Furthermore, byte-level memory writes are possible on most common architectures, including x86, x64, ARM, and MIPS. A *partial* pointer overwrite can be used to overwrite select bytes within a word. Partial pointer overwrites have been leveraged in previous exploits [8,19], however, in this work we leverage them in a more general attack technique, RelROP.

In this paper, we assume each memory page is 4 KB, and aligned on 4 KB boundaries. Therefore, the low-order 12 bits of each address represent the offset of the address within the page, while the high-order bits identify the page itself. We define a *memory paragraph* to be the subset of a page that is addressable by overwriting the low-order byte of a pointer. Thus, paragraphs are aligned $2^8 = 256$ byte regions of memory.

If virtual-memory randomization is applied, then the contents of each page are fixed, and can be determined offline by an attacker. Therefore, the memory paragraphs are also fixed, and the attacker can overwrite the low-order byte of an address to point to any gadget within the paragraph. This general concept is depicted in Fig. 1. The question marks denote that those bytes of the pointer are both *unknown* to the attacker (due to the presence of a leakage-resilient technique) and *uncorrupted* by the attacker. The low-order byte, however, which denotes an offset into the paragraph, are corrupted by the attacker by only overwriting a subset of the bits encoding the pointer. The corrupted pointer now points to a gadget within the paragraph, despite the presence of a leakage-resilient technique that protects pointers from disclosure. Note that the attacker-controlled pointer cannot point outside of the page without learning or guessing the value of randomized high-order bytes. Moreover, it cannot point to any other paragraph within the target page because even

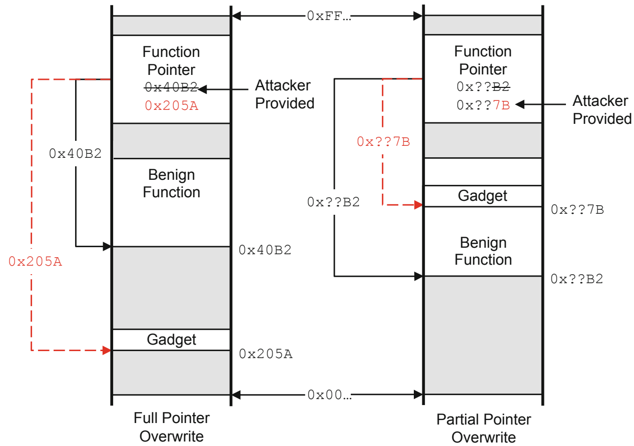

Fig. 1. Partial vs. full pointer overwrites

though bits 9–12 of the address are known to the attacker (from an attacker's local copy), they cannot be overwritten by byte-granularity memory-corruption.

At a high level, all that is required to carry out the attack is the ability to overwrite the low-order byte of the pointer that encodes a position within the pointed-to paragraph, while avoiding any corruption of the randomized higher-order bytes. This can be accomplished using a direct memory-write vulnerability (similar to CVE-2017-0106).[2] Such vulnerabilities arise from unchecked array offset references, for example.

4.2 RelROP Chaining

In order to construct a RelROP gadget chain, we leverage the layer of indirection afforded by the procedure linking table (PLT) and the global offset table (GOT). Each externally linked function, such as those in `libc`, is invoked via a `call` instruction to an absolute address within the PLT. The code within the PLT performs a lookup of the address of the called function within the GOT, and redirects control flow to that address. The GOT and PLT have two key features that enable RelROP chaining.

First, the GOT is in the data region, which is subject to neither the write protections of $W \oplus X$ nor to randomization. Thus, entries in the GOT are vulnerable to partial pointer overwrites. By corrupting GOT entries, the pointer can be offset relative to the function's intended entry point into an attacker-chosen memory region within the paragraph pointed to by that entry.

Second, the PLT is not part of the `.text`/`.code` section, and is therefore not randomized. It does contain code pages, however, so both $W \oplus X$ and leakage-resilience are in effect. Thus, the PLT itself cannot be directly leaked. However, the GOT contains pointers into the PLT in order to support lazy loading of library functions. This standard functionality allows function addresses to be resolved only on use, increasing the speed of program loading. However, it requires pointing un-initialized function pointers (*e.g.,* `_exit` should contain an entry back to its PLT entry) to stub code in the PLT, thus leaking its location.

With these capabilities, a series of pointers to functions in the PLT can be placed on the stack, similar to a standard ROP attack. When these pointers are dereferenced, they will be redirected via the corrupted GOT to attacker-chosen gadgets. This permits chaining of RelROP attacks.

5 RelROP Prevalence Analysis

RelROP attacks leverage GOT entries to address gadgets at a relative offset from that pointer's initial location. In order to investigate the prevalence of gadgets accessible at the paragraph level of granularity, we constructed an analysis tool and applied it to over 1,300 binaries, analyzing the libraries and functions that were dynamically linked by these binaries. In this analysis, we identify all gadgets that are accessible by partially overwriting the low-order byte of a GOT entry.

[2] Note that other vulnerability types could also be used. For example, buffer overflows (resp. underflows) could be used, in little-endian (resp. big-endian) architectures.

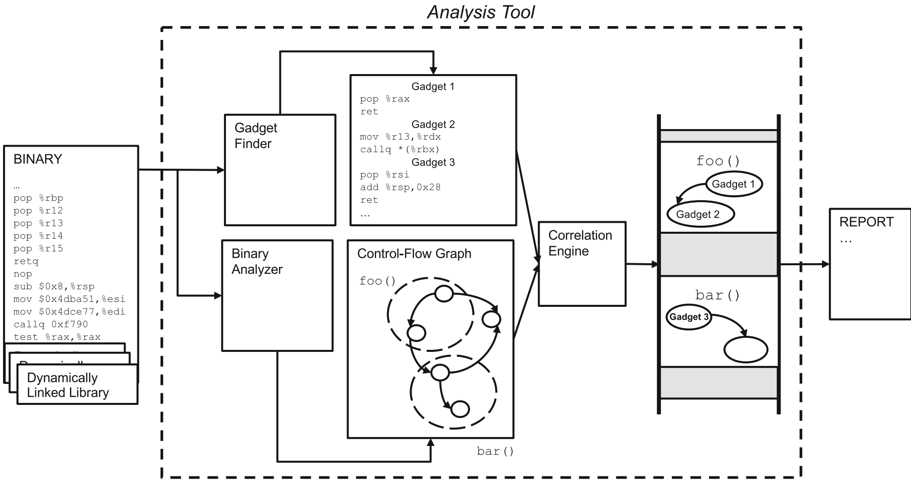

Fig. 2. RelROP gadget prevalence analysis tool architecture

5.1 Analysis-Tool Architecture

The high-level architecture of our analysis tool is depicted in Fig. 2. An input binary is processed in three phases.

First, we leverage `angr` [44], an open-source binary analysis framework, to identify all of the libraries that are linked to a given binary. Then, all conventional ROP gadgets are identified in all of these libraries using an off-the-shelf tool (these are filtered later). We chose to use the open-source tool `rp++` [3] for this purpose, with a search depth of 8 instructions (*i.e.,* each identified gadget is at most 8 instructions long).

Next, we use `angr` to identify all functions from libraries that are actually imported by the binary. That is, we *only* consider functions that actually appear in the binary's PLT, and are thus usable by RelROP. Finally, we use the function information from `angr` to identify all of the gadgets that can be accessed by overwriting the low-order byte of that function's GOT entry. Note that for each case, gadgets can be found within the function (*i.e.,* a positive offset) or within the memory *before* the function (*i.e.,* a negative offset). This is because the physical memory pages of these libraries must remain static during runtime. Thus, in the case of paragraph level randomization we consider every gadget within the memory paragraph (*e.g.,* if the function pointer is `0x11223344`, any gadget in the range `0x11223300`-`0x112233FF` is accessible).

5.2 Analysis of Real-World Binaries

In order to characterize how prevalent RelROP gadgets are, we ran our tool on every binary contained within the `/usr/bin` and `/usr/sbin` directories on a developer machine (Ubuntu 16.04), totaling 1,365 binaries with 577 dynamically

linked libraries. The results of this analysis are summarized in Table 1. In this table, the first column represents the major gadget classes, and the next two columns depict the percentage and total, respectively, of analyzed binaries that include a gadget of each class at the paragraph granularity. The percentage of binaries with such gadgets accessible through `libc` is also included alongside the results, as attacks using libc gadgets are more desirable because of their reusability across binaries. These results demonstrate that there are ample gadgets available via partial pointer overwriting even when the attacker is constrained to the gadgets within a single byte of a code pointer.

The results in Table 1 summarize raw metrics on the number of gadgets available, but do not directly address whether there are sufficient gadgets to carry out a RelROP attack. The next step in our evaluation is to identify the fraction of applications that have enough RelROP gadgets to carry out a more complete malicious payload, such as spawning a shell. Specifically, we consider an application vulnerable to a RelROP-spawned shell if it includes either a `mov` or `pop` gadget for all the registers needed for the `execve` syscall (*i.e.,* `rax`, `rdx`, `rsi`, and `rdi`), as well as a syscall gadget. Our analysis determined that *94.4% of the binaries we considered are vulnerable, and 91.4% are vulnerable if gadgets are restricted to libc only.* These results suggest that virtual memory randomization is not, on its own, sufficient to prevent RelROP attacks.

Table 1. Gadgets within paragraph of GOT entry

Gadget	Percentage of binaries with gadgets/libc portion	Total number of gadgets/libc portion
`pop rax`	80.8%/70.2%	64493/12196
`mov rax`	99.7%/99.7%	1118428/378268
`pop rbx`	99.7%/96.3%	2326697/550486
`mov rbx`	82.3%/69.0%	81541/21715
`pop rcx`	79.2%/63.4%	43827/14253
`mov rcx`	90.8%/83.7%	214140/81593
`pop rdx`	66.9%/46.7%	28827/11845
`mov rdx`	99.7%/99.7%	418448/151041
`pop rsi`	95.6%/92.2%	123090/20512
`mov rsi`	99.7%/99.7%	426279/96681
`pop rdi`	95.2%/91.6%	97963/22853
`mov rdi`	93.6%/86.9%	831198/189329
`syscall`	94.8%/93.5%	1067064/814589

We note that in practice, an application may have gadgets that affect all of the necessary registers, but chaining the gadgets together for a successful attack may not be feasible given other side effects present in the gadgets. Additionally, our results are predicated on the completeness of our gadget-analysis tool, and other gadget analyses may identify other gadgets. These results are thus presented as indicative of RelROP prevalence, but are not claimed to be comprehensive.

6 Real-World Exploit

For our real-word exploit, we selected our target based on disclosed CVEs and *not* the availability of gadgets, since our prevalence analysis had already shown that there were likely enough gadgets to construct an exploit payload. Our real-world exploit targets the popular PHP: Hypertext Preprocessor (PHP). Specifically we

Table 2. List of ROP gadgets identified within the entry paragraph of library functions used by PHP 7.0.0

Library	Function	Offset	Gadget
libc-2.23.so	inet_ntoa	0x47	`pop rax; mov rax,rbx; pop rdx; pop rbx; ret;`
libc-2.23.so	uname	0x05	`syscall;`
libicuuc.so.55.1	u_isISOControl_55	0x05	`pop rsi; setnbe dl; cmp edi,0x0000009F; setbe al; and eax,edx; ret;`
libicuuc.so.55.1	UnicodeString::doCompare	0x03	`pop rdi; or byte [rcx-0x0A],al; ret;`
libxml2.so.2.9.3	xmlParseBalancedChunkMemory	0x04	`pop rcx; add byte [rax],al; add byte [rsi+0x06],bh; ret;`

targeted PHP version 7.0.0, and leveraged a known format-string vulnerability (*i.e.,* CVE-2015-8617 [1]) as a proof-of-concept for both leaking and exploiting the GOT.

Note that because of the existence of $W \oplus X$, code regions cannot be written to and data regions cannot be executed. Moreover, because of the deployment of a leakage-resilient defense, code regions cannot be reliably read. As a result, we only assume a read/write capability to *data* pages of memory in our exploit.

6.1 Exploit Details

The goal of our exploit is to achieve control-flow hijacking while PHP is protected by a leakage-resilient defense using virtual-memory randomization up to and including page-level randomization (thus, we are restricted to gadgets within the paragraph of a function pointer). Since PHP is an interpreter, we assume that the attacker is permitted to execute their own malicious PHP file on a remote server, as is common on most hosting providers. To demonstrate a powerful attack, we design an exploit that invokes the `execve` system call to spawn a new shell. This provides the attacker with powerful remote control over the compromised machine with elevated privileges from that of the original PHP script. To accomplish this, we must find a `syscall`-instruction gadget and a set of gadgets to set the necessary argument registers (*i.e.,* `rax`, `rdi`, `rsi`, and `rdx`).

We applied the tool described in Sect. 5 to analyze, offline, a local copy of PHP to identify all of the gadgets that are contained within the entry paragraph (*i.e.,* the paragraph surrounding the pointer to a function's entry point) of every function that is imported by PHP. Note that we can craft our malicious PHP file to specifically call those functions that contain the required gadgets to ensure that the GOT will be populated before our exploit. Our attack is limited to only use gadgets that are contained within entry paragraphs (*i.e.,* the single-byte offset from the function-entry point), which is encoded in the GOT. This constant offset can be added by overwriting only the low-order byte in the GOT entry, which is not affected by randomization at the page-level or coarser granularity. The gadgets identified by our tool are shown in Table 2.

It is worth noting that our `pop rdi` gadget depends on the value of `rcx-0x0A` being a valid and writable memory region. Similarly, our `pop rcx` gadget requires

`rax` and `rsi+0x06` to be writable. Fortunately, we have both `pop rax` and `pop rsi` gadgets that we can use to set these values to known locations in the GOT, which we know to be writable. We can then similarly set `rcx` to a known GOT address to achieve a complete payload.

In traditional ROP attacks, the attacker places the absolute address of the gadgets directly on the stack in order to execute them in the payload. However, in RelROP, we are working with the constraint of virtual-memory randomization and leakage resilience, thus RelROP places the *PLT* addresses on the stack, which will be automatically resolved to our corrupted GOT entries.

To set up the exploit, we leverage the fact that the `.data` segment, including the GOT, is not randomized and is always at a fixed memory location. In the case where this is not true, we could use our memory-read vulnerability (*i.e.,* our format-string vulnerability) to leak the location of the GOT. Given any GOT address, we can trivially calculate the base address, and therefore the address of the functions containing our gadgets, as the order of the GOT entries do not change. This same format string can be leveraged to read the contents of the GOT to obtain the base address of the PLT, as unresolved functions will store pointers to their PLT entry due to lazy binding of library functions.

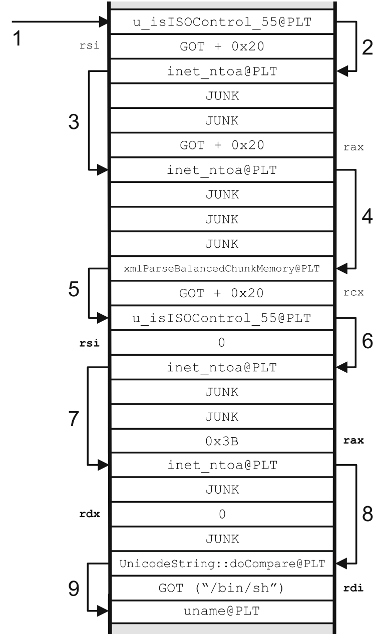

Fig. 3. The stack during exploitation

At this point, we have enough information to modify the GOT entries and build the set of values that need to be placed on the stack when the exploit begins executing.

Next, we modify the lower-order bits of the GOT entries for `gethostbyname`, `php_uname`, `intltz_to_date_time_zone`, `IntlChar::isISOControl`, and `DOMDocument::appendXML` (PHP functions that call the functions listed in Table 2) by partially overwriting each entry with the offset of the gadget located in each respective function.

We start by using an assumed arbitrary-write stack-corruption vulnerability to place the proper values on the stack and point the return address to the first gadget. The stack is setup similarly to a traditional ROP payload, containing data that will end up in registers, and addresses of gadgets to be executed. Instead of using the absolute address of the gadgets, however, we use the address of the PLT entries of the functions containing the gadgets. It is important to emphasize that we know the addresses in the PLT from pointers in the GOT used for lazy binding, not from a leakage of the PLT that is prevented by the leakage-resilient defense. The stack during our RelROP attack is shown in Fig. 3. The full exploit is shown in Fig. 4.

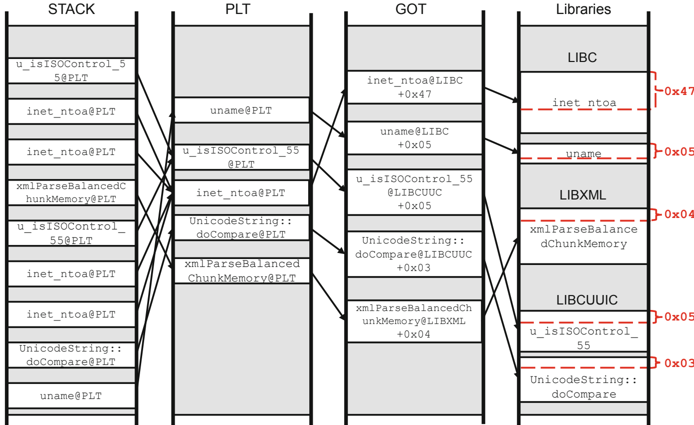

Fig. 4. PHP RelROP exploit

7 Impact on Defenses

In this section, we consider the impact of RelROP attacks on two classes of defenses. Randomization-focused defenses are those whose primary mechanism for mitigating attacks is (re)-randomization of memory at a specific level of granularity. Randomization-dependent defenses are those that require fine-grained memory randomization, but whose primary contribution is orthogonal to randomization (*e.g.,* execute-only memory).

7.1 Randomization-Focused Defenses

Table 3 summarizes the impact of RelROP on leakage-resilient defenses. These include both leakage-resilient defenses that rely on memory re-randomization, and fine-grained randomization mechanisms that may be used by leakage-resilient defenses that are dependent on a fine-grained randomizer. The table also indicates whether the requirements to conduct a RelROP attack are satisfied. We require a GOT to exist and not be *additionally* protected, and that the target be protected by either virtual-memory randomization, or physical-memory randomization that does not extend to shared libraries.

TASR is susceptible RelROP attacks. It is a leakage-resilient defense that re-randomizes code at the library level. Since the GOT is in the data region, it is not randomized by TASR. Re-randomization is applied on every read/write pair to mitigate the effects of memory disclosures. While its coverage does extend to shared libraries, it is implemented via virtual-memory randomization, and is therefore susceptible to RelROP attacks given the analysis presented in Sect. 5.

Remix is a leakage-resilient defense that periodically permutes the basic-block ordering within functions. This necessitates physical memory copies and code patching to ensure that direct jumps point to the correct target. Consequently,

Table 3. Susceptibility of leakage-resilient techniques to RelROP

Defense name	Granularity	Randomization	Unprotected GOT	Unprotected libraries	RelROP
Leakage resilience through memory re-randomization					
TASR [7]	Library	Virtual	Yes	No	Yes
Shuffler [53]	Function	Physical	No	No	No
Remix [12]	Basic Block	Physical	Yes	Yes	Yes
Memory randomization					
Oxymoron [5]	Page	Virtual	No	No	No
Binary Stirring [51]	Basic Block	Physical	Yes	Yes	Yes
ILR [26]	Instruction	Physical	Yes	Yes	Yes

Remix does not protect shared libraries. Since RelROP attacks use only gadgets in shared libraries, Remix is susceptible to RelROP.

Binary Stirring [51] is a load-time basic-block-level randomization technique. It relies on load-time patching of the binary to redirect direct jumps to randomly determined basic-block locations. Consequently, shared libraries are not randomized and can be leveraged to conduct RelROP attacks.

ILR [26] uses process-level virtualization to perform instruction-level randomization. Since this does not extend across processes, shared libraries are not protected and RelROP attacks can bypass it.

Oxymoron [5] randomizes code on the page level, as well as replacing function pointers with trampolines into a protected, GOT-like memory region. This region is isolated via memory segmentation and segment registers. This prevents RelROP attacks due to the inability to partially corrupt function pointers in the GOT. Unfortunately, attacks against it have already been demonstrated [16] and memory segmentation is largely unsupported in 64-bit architectures.

Shuffler is a leakage-resilient defense that is not susceptible to RelROP attacks, as it removes the GOT and relies purely on direct calls to libraries that are statically linked at load time. It periodically re-randomizes code at the function level at a configurable interval. Since functions may be smaller than pages, this randomization requires physical memory copying. This necessitates statically linking shared libraries. Due to the way Shuffler implements re-randomization, the size of each process' code image (including all libraries) is approximately doubled. As a result, the memory overhead on a multi-process system may be prohibitive. Shuffler also requires a dedicated per-process thread to asynchronously perform physical memory copies, which may impact cache and memory performance. Unfortunately, no analysis is provided as to the performance of Shuffler in a multi-process environment, so the true overhead is difficult to estimate.

7.2 Randomization-Dependent Leakage-Resilient Defenses

The defenses considered in this section rely on the existence of a fine-grained randomization mechanism, but their primary contribution is an orthogonal approach to leakage resilience. Since "fine-grained randomization" is often underspecified, the effect of RelROP attacks on each defense cannot be empirically evaluated. Thus, we instead consider whether the GOT/PLT is additionally protected or other implementation details disrupt RelROP attacks.

Multivariant Execution. Multivariant-execution defenses, such as Isomeron [16], are designed to disrupt ROP and JIT-ROP attacks by probabilistically switching program execution among two or more replicas of code, each with different memory layouts. Isomeron specifically applies "fine-grained" code randomization to one of two replicas, and leaves the other unmodified. Execution switches uniformly at random between each replica at the function-call granularity. This disrupts code-reuse attacks that rely on absolute jumps to memory addresses, as the location of gadgets may change at every gadget invocation. However, if the underlying code randomization is virtual-memory randomization, it does not disrupt RelROP attacks. GOT entries in Isomeron are resolved prior to diversification, and Isomeron adds a constant offset to the result if it elects to change the replica being run. Since RelROP attacks corrupt GOT entries prior to this calculation, they are "fixed" by Isomeron to point to the correct replica. If physical-memory randomization is applied to either replica, the partially corrupted pointer would point to a different location in each replica, and therefore the attack would not succeed.

Destructive Code Reads and Execute-Only Memory. Techniques that implement destructive code reads [36,50,52] aim to prevent code-reuse attacks that rely on direct memory disclosure. While all code pages can be both read and executed (in contrast to execute-only memory), attempting to execute code that has previously been read will trigger an error. In response to inference attacks that allow implicit disclosure of code by reading adjacent bytes [46], this approach has recently been combined with semantic-preserving binary re-randomization [36]. Execute-only-memory defenses [4,22] aim to stop the same class of threats as defenses that implement destructive code reads. Rather than destroying code that is read, however, execute-only defenses cause a memory-permission violation at any attempt to read executable memory.

Both of these defense classes rely on the necessity of an attacker reading *code* pages prior to executing that code. However, RelROP attacks rely entirely on reading *data* pages and corrupting code pointers without first disclosing that code (or its address). Only the GOT itself needs to be read, which, as data, does not trigger destruction. Therefore, if virtual-memory randomization is used, then partial pointer overwriting can be used to corrupt code pointers to known gadgets within the containing code paragraph. However, if physical-memory randomization is applied, then the byte value needed for the partial overwrite cannot be determined without first disclosing the randomization, and thus physical-memory randomization would prevent a RelROP attack.

Code-Pointer Protection. Another approach to preventing code-reuse attacks is to protect all pointers to code from disclosure or corruption. Pointguard [13] encrypts pointers and decrypts them just prior to use via a register-stored key. ASLR-Guard [35] uses a combination of encryption and protected lookup tables to hide the value of function pointers. Readactor [14,15] combines execute-only memory, fine-grained code randomization, register randomization, PLT randomization, and replacement of function pointers with trampolines into a protected lookup table. Notably, however, Readactor has been shown vulnerable to profiling-based attacks [41].

Encrypting or otherwise protecting all bytes of function pointers prevents partial overwrites, as low-order bits are no longer vulnerable. In addition, use of trampolines into lookup tables decouples the pointer value from any gadgets near its eventual target, thus making relative-address attacks only able to (at best) change the index into the lookup table. If table randomization and booby traps are used, as in Readactor, even this capability is removed. Thus, code-pointer protection techniques are effective in countering RelROP attacks.

8 Discussion

Physical-memory randomization at the granularity of instruction or basic-blocks, applied ubiquitously to the binary and its linked libraries would not be vulnerable to the RelROP attack described earlier. However, such a technique faces a number of practicality challenges. Furthermore, subsequent design decisions to address those challenges themselves come with security/practicality implications. All of these challenges arise from dealing with shared physical memory pages, such as those in linked libraries. In this section, we first discuss the practical challenges of physical-memory randomization, then we discuss other possible RelROP mitigations.

8.1 Implications of Physical-Memory Randomization

Cross-Process Disclosures. Many physical-memory randomization defenses (see [34] for an overview) apply randomization at compile time, by, for example, inserting NOPs to change relative distances between instructions. These one-time randomization approaches suffer from the fact that a memory disclosure in *any* process using a shared code page (*e.g.,* `libc` pages) allows the attacker to derandomize that page in *all* processes using that code page. Thus, leakage-resilient defenses must be applied to every process that links shared libraries.

Shared-Library Synchronization. Physical-memory randomization that takes place at load- or run-time must deal with the fact that multiple processes executing code from shared libraries do not synchronize their accesses, as these pages are traditionally read-only. This becomes problematic when attempting to move that code to another physical memory region. Each process may have stack/heap pointers to different regions of the shared library (especially if library functions

call each other), and have instruction pointers at different addresses within that library. All of these pointers must be adjusted to point to the library's new location in a way that is transparent to each running application.

Shuffler [53] addresses the issue by statically linking all libraries into a process image at load time, and maintaining two copies of the process binary and libraries. One copy is active and used for execution, and the other is asynchronously re-randomized by a dedicated thread. When the copy is complete, execution shifts to the new version and re-randomization is applied to the other copy. Unfortunately, this means that if n processes are executing on a system, there are $2n$ copies of `libc`, $2n$ copies of each application binary, and up to $2n$ copies of other shared libraries. Remix [12], Binary Stirring [51], and ILR [26] address this issue by simply not protecting shared libraries, and limiting themselves to the unshared physical pages corresponding to the main application binary. As shown in Sect. 5, however, this still provides ample attack surface to create a malicious payload. In fact, most valuable gadgets, such as those capable of invoking a system call, are found in `libc` and not the binary itself.

Memory Thrashing. Runtime re-randomization based on physical-memory randomization, such as Remix [12] and Shuffler [53], periodically perform physical memory copies in order to relocate code regions. This interferes with the performance of the cache and memory subsystem due to large-scale invalidation of cache lines, and additional memory traffic. Depending on the rate at which re-randomization is performed, memory thrashing can become a significant source of overhead. A study of cache and memory performance observed such cache and memory contention can result in slowdowns of a factor of up to 2.5x [32].

8.2 RELRO

A defensive feature in some operating systems called Relocation Read-Only (or RELRO) is sometimes used to protect GOT. Partial RELRO forces GOT to come before BSS, preventing some types of buffer overflows on global variables. Full RELRO marks the entire GOT as read-only.

While partial RELRO has no impact on RelROP, full RELRO breaks it. However, full RELRO has several performance tradeoffs, and is not commonly deployed in practice. A recent study shows that as low as 3% of binaries are protected with full RELRO [48]. There are a few reasons for this. Full RELRO requires all symbols to be resolved at load time, which significantly slows down program startup. Full RELRO is also not a default option in GCC (partial RELRO is). Many Linux distros also do not have RELRO, such a RHEL v6 (and earlier), which will be actively supported until 2021.

9 Related Work

Our work mainly relates to memory-corruption vulnerabilities and mitigation thereof. The literature in these areas is vast. We refer the interested reader to

the relevant surveys [10,34,49] and focus on closely related work. Since we have already discussed may related efforts in the context of our attack, we limit the work referenced in this section to the remaining closely related ones.

In a concurrent work with ours, a similar attack, PIROP [24], also uses partial pointer overwrites to bypass leakage-resilient defenses. However, PIROP's approach is significantly different from ours in the following aspects. First, PIROP is based on the concept of memory massaging, in which a carefully chosen set of inputs causes the program to place code pointers on the stack. These are then adjusted via partial pointer overwrites. This approach is probabilistic under fine-grained randomization, with probability of success decreasing as the required number of gadgets increases. RelROP attacks, conversely, are deterministic and can scale to arbitrary payload sizes. Second, it is unclear how well PIROP attacks generalize or could be automated. Each proof of concept exploit presented in that work requires study and use of application-specific execution semantics. RelROP attacks only require knowledge of the target binary's GOT. Third, PIROP attacks are only able to bypass memory re-randomization defenses if they are restricted to live pointers that are actively being tracked by the re-randomizer. They cannot rely on stale pointers, such as those remaining from old stack frames whose associated function has already returned. RelROP attacks bypass any virtual memory re-randomization technique. Fourth, PIROP's evaluation focuses on the amount of entropy provided by various existing defenses. Since RelROP attacks are deterministic, this does not apply to our technique. We instead analyze the tradeoffs between virtual and physical memory randomization, and their implications for practical leakage-resilient defenses.

There are also a large number of randomization-based techniques proposed in the literature that perform compile-time [28,29,33], load-time [17,26], or runtime [27,37] randomization. It has been shown that information-leakage attacks of various types, including direct memory disclosures [47], timing-based and fault-based side-channel attacks [8,42], script-based leaks [45], indirect pointer leaks [16], profiling attacks [41], and cache-based side-channel attacks [25], can be used to bypass randomization-based defenses. Other orthogonal attacks against many leakage-resilient defenses have also been studied, the details of which are beyond our scope [6,9,16,20,40,41,45,46].

Control flow integrity (CFI) and all of its variants [10] are another class of memory corruption defenses that are orthogonal to and not impacted by RelROP. They are, however, vulnerable to attacks on the imprecisions of the control flow graph [21].

10 Conclusion

In this paper, we analyzed the security and practicality of memory-randomization mechanisms supporting leakage-resilient defenses. We illustrated an attack, RelROP, that bypasses page-level or coarser virtual-memory randomization via partial overwriting of code pointers. We analyzed the prevalence of RelROP gadgets in popular code bases, and built a proof-of-concept exploit

against PHP 7.0.0. In addition, we enumerated the challenges associated with practical deployment of physical-memory randomization defenses that arise from protecting shared memory objects (*e.g.,* shared libraries). Our findings indicate that additional research is needed to design efficient and effective leakage-resilient memory-protection techniques.

References

1. CVE-2015-8617. "Available from MITRE, CVE-ID CVE-2015-8617" (2015). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8617
2. Threat LandScape Report Q2 2017. Fortinet (2017). https://www.fortinet.com/content/dam/fortinet/assets/threat-reports/Fortinet-Threat-Report-Q2-2017.pdf
3. 0vercl0k: rp++, April 2017. https://github.com/0vercl0k/rp
4. Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., Pewny, J.: You can run but you can't read: preventing disclosure exploits in executable code. In: ACM Conference on Computer and Communications Security. CCS (2014)
5. Backes, M., Nürnberger, S.: Oxymoron: making fine-grained memory randomization practical by allowing code sharing. In: 23rd USENIX Security Symposium. USENIX Sec (2014)
6. Barresi, A., Razavi, K., Payer, M., Gross, T.R.: CAIN: silently breaking ASLR in the cloud. In: 9th USENIX Security Symposium. WOOT 2015 (2015)
7. Bigelow, D., Hobson, T., Rudd, R., Streilein, W., Okhravi, H.: Timely rerandomization for mitigating memory disclosures. In: ACM Conference on Computer and Communications Security. CCS (2015)
8. Bittau, A., Belay, A., Mashtizadeh, A.J., Mazières, D., Boneh, D.: Hacking blind. In: 35th IEEE Symposium on Security and Privacy. S&P (2014)
9. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup est machina: Memory deduplication as an advanced exploitation vector. In: 37th IEEE Symposium on Security and Privacy (2016)
10. Burow, N., et al.: Control-flow integrity: precision, security, and performance. ACM Comput. Surv. **50**(1), 16:1–16:33 (2017)
11. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: ACM Conference on Computer and Communications Security. CCS (2010)
12. Chen, Y., Wang, Z., Whalley, D., Lu, L.: Remix: on-demand live randomization. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 50–61. ACM (2016)
13. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguard: protecting pointers from buffer overflow vulnerabilities. In: 12th USENIX Security Symposium. USENIX Sec (2003)
14. Crane, S., et al.: Readactor: practical code randomization resilient to memory disclosure. In: 36th IEEE Symposium on Security and Privacy. S&P (2015)
15. Crane, S., et al.: It's a TRaP: table randomization and protection against function-reuse attacks. In: ACM Conference on Computer and Communications Security. CCS (2015)
16. Davi, L., Liebchen, C., Sadeghi, A.R., Snow, K.Z., Monrose, F.: Isomeron: code randomization resilient to (Just-In-Time) return-oriented programming. In: 22nd Annual Network and Distributed System Security Symposium. NDSS (2015)

17. Davi, L.V., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In: ASIACCS, pp. 299–310 (2013)
18. De Sutter, B., Anckaert, B., Geiregat, J., Chanet, D., De Bosschere, K.: Instruction set limitation in support of software diversity. In: Lee, P.J., Cheon, J.H. (eds.) ICISC 2008. LNCS, vol. 5461, pp. 152–165. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00730-9_10
19. Durden, T.: Bypassing PaX ASLR protection (2002). http://www.phrack.org/issues.html?issue=59&id=9
20. Evans, I., et al.: Missing the point(er): on the effectiveness of code pointer integrity. In: 36th IEEE Symposium on Security and Privacy. S&P (2015)
21. Evans, I., et al.: Control jujutsu: on the weaknesses of fine-grained control flow integrity. In: ACM Conference on Computer and Communications Security. CCS (2015)
22. Gionta, J., Enck, W., Ning, P.: HideM: protecting the contents of userspace memory in the face of disclosure vulnerabilities. In: 5th ACM Conference on Data and Application Security and Privacy. CODASPY (2015)
23. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: 21st USENIX Security Symposium. USENIX Sec (2012)
24. Göktas, E., et al.: Position-independent code reuse: on the effectiveness of ASLR in the absence of information disclosure. In: IEEE EuroS&P (2018)
25. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the line: Practical cache attacks on the MMU. NDSS, February 2017 (2017)
26. Hiser, J., Nguyen, A; Co, M., Hall, M., Davidson, J.: ILR: Where'd my gadgets go. In: 33rd IEEE Symposium on Security and Privacy. S&P (2012)
27. Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Librando: transparent code randomization for just-in-time compilers. In: ACM Conference on Computer & Communications security, pp. 993–1004 (2013)
28. Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., Franz, M.: Profile-guided automated software diversity. In: International Symposium on Code Generation and Optimization (CGO), pp. 1–11. IEEE (2013)
29. Jackson, T., et al.: Compiler-generated software diversity. In: Moving Target Defense. Advances in Information Security (2011)
30. Jackson, T., Homescu, A., Crane, S., Larsen, P., Brunthaler, S., Franz, M.: Diversifying the software stack using randomized NOP insertion. In: Moving Target Defense. Advances in Information Security (2013)
31. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In: 22nd Annual Computer Security Applications Conference. ACSAC (2006)
32. Kim, N., Ward, B.C., Chisholm, M., Anderson, J.H., Smith, F.D.: Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. Real-Time Syst. **53**(5), 709–759 (2017)
33. Koo, H., Chen, Y., Lu, L., Kemerlis, V.P., Polychronakis, M.: Compiler-assisted code randomization. In: IEEE Symposium on Security & Privacy (SP) (2018)
34. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: 35th IEEE Symposium on Security and Privacy. S&P (2014)
35. Lu, K., Song, C., Lee, B., Chung, S.P., Kim, T., Lee, W.: ASLR-Guard: stopping address space leakage for code reuse attacks. In: ACM Conference on Computer and Communications Security. CCS (2015)

36. Morton, M., Koo, H., Li, F., Snow, K.Z., Polychronakis, M., Monrose, F.: Defeating zombie gadgets by re-randomizing code upon disclosure. In: International Symposium on Engineering Secure Software and Systems, pp. 143–160 (2017)
37. Novark, G., Berger, E.D.: Dieharder: securing the heap. In: ACM Conference on Computer and Communications Security. CCS, pp. 573–584 (2010)
38. One, A.: Smashing the stack for fun and profit. Phrack Mag. **7**, 14–16 (1996)
39. PaX: PaX address space layout randomization (2003)
40. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip feng shui: hammering a needle in the software stack. In: 25th USENIX Security Symposium. USENIX Sec (2016)
41. Rudd, R., et al.: Address-oblivious code reuse: on the effectiveness of leakage resilient diversity. In: Proceedings of the Network and Distributed System Security Symposium. NDSS 2017, February 2017
42. Seibert, J., Okhravi, H., Söderström, E.: Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: ACM Conference on Computer and Communications Security. CCS (2014)
43. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security. CCS (2007)
44. Shoshitaishvili, Y., et al.: SoK: (State of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
45. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: 34th IEEE Symposium on Security and Privacy. S&P (2013)
46. Snow, K.Z., Rogowski, R., Werner, J., Koo, H., Monrose, F., Polychronakis, M.: Return to the zombie gadgets: undermining destructive code reads via code inference attacks. In: 37th IEEE Symposium on Security and Privacy (2016)
47. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: 2nd European Workshop on System Security. EUROSEC (2009)
48. Saito, T., Yokoyama, M., Sugawara, S., Suzaki, K.: Safe trans loader: mitigation and prevention of memory corruption attacks for released binaries. In: Inomata, A., Yasuda, K. (eds.) IWSEC 2018. LNCS, vol. 11049, pp. 68–83. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-97916-8_5
49. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: eternal war in memory. In: Proceedings of IEEE Symposium on Security and Privacy (2013)
50. Tang, A., Sethumadhavan, S., Stolfo, S.: Heisenbyte: thwarting memory disclosure attacks using destructive code reads. In: ACM Conference on Computer and Communications Security. CCS (2015)
51. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: ACM Conference on Computer and Communications Security. CCS (2012)
52. Werner, J., et al.: No-execute-after-read: preventing code disclosure in commodity software. In: 11th ACM Symposium on Information, Computer and Communications Security. ASIACCS (2016)
53. Williams-King, D., et al.: Shuffler: fast and deployable continuous code rerandomization. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, pp. 367–382 (2016)