On the Weaknesses of Function Table Randomization

Moritz Contag, Robert Gawlik, Andre Pawlowski, and Thorsten Holz

Horst Görtz Institute (HGI), Ruhr-Universität Bochum, Germany

Abstract. Latest defenses against code-reuse attacks focus on information hiding and randomization as important building blocks. The main idea is that an attacker is not able to find the position of the code she wants to reuse, hence thwarting successful attacks. Current state-of-the-art defenses achieve this by employing concepts such as execute-only memory combined with booby traps.

In this paper, we show that an attacker is able to abuse symbol metadata to gain valuable information about the address space. In particular, an attacker can mimic dynamic loading and manually resolve symbol addresses. We show that this is a powerful attack vector inherent to many applications using symbol resolving at runtime, an ubiquitous concept in today's systems. More importantly, we utilize this approach to resolve and reuse functions otherwise unavailable to an attacker due to *function table randomization*. To confirm the practical impact of this attack vector, we demonstrate how dynamic loading can be exploited to bypass Readactor++, the state-of-the-art defense against code-reuse attacks, despite its use of booby traps and virtual function table (*vtable*) randomization. Furthermore, we present a novel approach to protect symbol metadata to defend against such attacks. Our defense, called Symtegrity, is able to safeguard symbols from an attacker, whilst preserving functionality provided by the loader. It is both orthogonal to existing defenses and applicable to arbitrary binary executables. Empirical evaluation results show that our approach has an overhead of roughly 8% during application startup. At runtime, however, no noticeable performance impact is measured, as evident from both browser and SPEC benchmarks.

1 Introduction

The continuous arms race between more sophisticated attacks and subsequent defense mechanisms has led to several new primitives both sides can make use of. More fine-grained ASLR [16] has been tackled by just-in-time ROP (JIT-ROP), which discloses memory pages at runtime and builds a ROP sequence on the fly [32]. In turn, several exploit mitigation systems have attempted to prevent such disclosure using *information hiding*. Most notably, *XnR* [3] and *HideM* [15] provide *execute-only* memory to mitigate *direct* code disclosure. *Heisenbyte* [33] also distinguishes memory access patterns, but implements *destructive code reads* instead of restricting access per se. Thus, it ensures that information read by an adversary cannot be directly used in her exploit. *TASR* [4], on the other hand, is a randomization-based approach, which indirectly prevents an attacker from using the information she obtained before re-randomization. Finally, the *Readactor* system employs multiple techniques (namely, *execute-only* memory and trampolines) in order to overcome both *direct* and *indirect* memory disclosure attacks [10].

Nonetheless, any form of fine-grained randomization or diversification is inherently limited when aiming to prevent attacks based on *function reuse*. While an attacker will be limited in finding gadgets ending on a return, which reside at any offset within a function, she can resort to building an exploit chain only consisting of gadgets starting at a function beginning. The underlying idea is that it is still possible to build a payload this way and, what is more, that there are more sources leaking function addresses. Known function-reuse attacks [25, 30] may obtain function addresses from functions imported from shared objects or virtual function tables as used in C++ binaries, respectively. This has led to the development of the state-of-the-art defense Readactor++ [11], which introduces booby traps [9] and function table randomization on top of Readactor in order to counter advanced function-reuse attacks. In particular, it prevents an attacker from re-using functions at virtual function callsites in case of an information leak. We consider Readactor++ to be the most complete mitigation to date.

On a more general note, any code-reuse attack requires an attacker to harvest pointers to code sequences of her interest in order to perform the desired computation. However, depending on the application layout as well as defenses present, not all interesting code sequences may be accessible via a pointer exposed by the application itself. Hence, an attacker might be able to obtain the required information by traversing the data structures present in the program's address space to obtain further pointers. Especially, the *symbol table* in Linux and the *export table* in Windows applications provide valuable information for an attacker. The symbols contained in this structure may refer to both functions and data, whereas arguably, the former is of higher interest for an attacker. As the loader parses the symbol table to serve symbol requests at runtime, it is naturally mapped into the application's address space. In Linux, symbol requests may be dispatched due to the first attempt to execute a lazily-bound function imported from another module (*dynamic linking*) or even explicitly upon a programmer's request (*dynamic loading*). Microsoft Windows uses a similar concept.

In this paper, we demonstrate that dynamic loading represents an attack surface not considered before in detail. If an attacker is equipped with capabilities to read from the symbol table, she is able to mimic the loader's symbol-resolving facilities and obtain function pointers of her choice exported by the module in question. Consequently, dynamic loading can be considered an Achilles' heel inherent to most applications: while strictly required in many practical scenarios, we show that it represents a powerful attack vector against many defenses. More specifically, we discuss the attack vector induced by dynamic loading in general, along with the required background. To demonstrate the practical impact, we show how the attack vector can be used to successfully exploit web browsers protected by the state-of-the-art exploit-mitigation system Readactor++. Along the way, we also demonstrate how booby traps inserted into the vtables are ineffective against *vtable crafting* (i. e., the usage of fake vtables) attacks on Linux. To ascertain feasibility on Windows the attack has also been implemented against Internet Explorer successfully, but had to be omitted due to space limitations.

To counter this type of attack, we discuss potential defenses and propose Symtegrity. In contrast, our defense leverages execute-only memory for hiding information about symbols: we replace readable symbol metadata with references to so-called *oracles* that return the symbol address when executed. The oracles are protected via execute-only memory and booby traps, thus an attacker is not able to disclose them (even in the presence of an arbitrary read/write primitive). To demonstrate the practical feasibility of the approach, we implemented a prototype of this defense for binary executables in a tool called Symtegrity. While our defense induces a start-up overhead of around 8% for the Chromium browser, the overhead during runtime is negligible in several browser benchmarks and SPEC. To foster research on this topic, we make our implementation freely available at `https://github.com/RUB-SysSec/symtegrity`.

In summary, we make the following contributions in this paper:

- **Dynamic Loading as an Attack Vector.** We show that dynamic loading is a potential attack vector inherent to many applications and has to be considered in the design of modern defenses.
- **Overcoming function table randomization and boopy traps.** We demonstrate a novel attack against the Readactor++ system. Our generic bypass is based on information used to implement dynamic loading and circumvents protective mechanisms present in Readactor++, such as booby traps and function table randomization.
- **Legacy-compatible, light-weight defense.** We propose a robust defense mechanism to mitigate the attack vector. It induces low overhead and can be added to arbitrary binary executables. Additionally, it is orthogonal to most proposed defenses.

2 Technical Background

2.1 Dynamic Loading

The linking process at build time, as well as the loading process at run time, both allow for a variety of different approaches. In the following, we provide an overview of both aspects and discuss why dynamic linking using lazy binding and dynamic loading are the preferred approaches in practice.

Static linking describes the process of resolving symbols imported by a binary object at compile time. The code or data belonging to the declared symbol is copied verbatim into the object that uses the symbol. In contrast, *dynamic linking* defers this process to runtime. Instead of copying the corresponding data into the object that *uses* the symbol, the data lies in the object *declaring* it, commonly called *shared object* or *shared library.* If a symbol is used for the first time (e. g., by calling an exported function), its address is resolved and cached for further use (called *lazy binding*). Although it is possible to resolve the addresses of all symbols during load time (so-called *eager binding*), it is not used per default because it significantly slows down the loading process.

Dynamic linking using lazy binding is in practice the de-facto standard for multiple reasons. For one, startup speed increases considerably, as symbols are merely loaded on demand, and only necessary shared libraries are loaded to improve responsiveness. Furthermore, when the same shared libraries are used by multiple processes, they are only copied once into physical memory and shared by these processes. In addition, dynamic linking has security implications: Delaying symbol resolution to run time—as opposed to compile time—results in more modular applications. Hence, if one component of an application is affected by a security-critical vulnerability, this component can easily be updated without having to recompile and redistribute the whole application. *Dynamic loading* provides the programmer with capabilities comparable to those used by the dynamic linker. Namely, he can load or unload shared objects into process memory and resolve symbols at runtime. This approach is mostly taken if, due to program logic, the choice of shared object is dependent on runtime state and cannot be made at link-time. For example, an optional feature in a shared object may only be loaded if said object file actually exists on disk. If the object is missing, the feature it provides is not available, but the application as a whole remains usable. To implement dynamic loading, the *glibc* standard library exposes, amongst others, the APIs `dlopen` and `dlclose` for loading and unloading objects, respectively. Furthermore, *glibc* provides APIs such as `dlsym` for resolving the address of a symbol within an object. Every object that exposes symbols contains a *symbol table* describing metadata associated with a symbol. `dlsym` parses said table upon looking up a symbol, which is why the information has to be loaded into (readable) memory.

2.2 Execute-only Memory and Booby Traps

Execute-only memory is a technique that allows the operating system to protect memory pages such that they are neither readable nor writable, but still executable. This enables a plethora of use cases. Most notably, this concept goes well with schemes relying on *information hiding* (e. g., of *safe regions* [19]). Discovery of a hidden page does not immediately lead to disclosure of the page's content to the attacker. Consequently, execute-only memory protects against *direct* memory disclosure as used in the just-in-time ROP (JIT-ROP) attack [32].

Booby traps, a concept proposed by Crane et al. [9], are a mechanism to actively detect and respond to attacks against a given application. The main idea of booby traps is as follows: in a diversified application, code sequences—the actual booby traps—are added that trigger an active response, such as terminating the program or generating an alert. In a regular program run, these code snippets lie dormant and do not interfere with the normal program execution. However, if an attacker blindly executes a memory location, such as an entry in a randomized vtable, chances are high that she will hit a booby trap early on.

This can be the case, for example, if the memory layout differs from what the attacker expects due to diversification.

2.3 Readactor++ Overview

Readactor [10] and its follow-up, Readactor++ [11], are state-of-the-art exploit mitigation systems. Due to space reasons, we can only briefly present the underlying approach, for a more detailed overview we refer to the corresponding papers and our technical report [8]. Readactor is a source-based solution and consists of several components. Its compiler applies fine-grained code diversification such that an attacker cannot make assumptions about the application's memory layout. Further, code and data are separated. This enables execute-only memory, which is implemented using extended page tables (EPT). Consequently, *direct* memory disclosure (as used in, e. g., JIT-ROP) is mitigated. Further, code pointers are hidden by replacing direct code references with *trampolines* lying in execute-only memory. This mitigates *indirect* memory disclosure, which leaks code layout via code pointers stored on the stack or heap (e. g., return addresses). The attacker can now merely leak the addresses of the trampolines. Readactor++ improves the system in order to mitigate function reuse attacks such as RILC [25] and COOP [30]. The core insight is that knowing the layout of function tables such as the Procedure Linkage Table (PLT) or virtual function tables (vtables) in C++ applications gives an attacker enough information to instantiate function-reuse attacks. Thus, Readactor++ chooses to *randomize* entries within both kinds of tables and again uses trampolines to *hide* its contents. Blind probing of table entries is mitigated by inserting booby traps in both PLT and vtables.

2.4 Crash Resistance

Recently introduced attack primitives abused either the ability of programs to restart or their ability to absorb critical access violations [13, 14, 31]. In both client and server applications, such primitives were used to safely probe the program's address space. Hence, the process does not terminate or automatically restarts if an address is queried that is either unmapped or not equipped with read permissions. Consequently, such primitives can be used to attack any form of *information hiding*: By simply scanning the whole address space, pages containing sensitive information (e. g., shadow stacks, process metadata, or encryption keys) can be unveiled, although no direct references to these sensitive regions exist. Especially probing the complete address space of complex applications, such as web browsers, is achievable within *one single* process, as it may survive erroneous memory accesses [14]. We term this type of crash resistance *intra-process crash resistance*. Scanning for hidden information in network services is possible within several processes, as each new request spawns a new server process [13, 31]. Each of these processes may terminate abnormally without terminating the program. To distinguish this case (i. e., multiple processes are used to enable crash resistance), we use the term *inter-process crash resistance*.

3 Caveats of Dynamic Loading

3.1 Attacker Model

For the rest of the paper, we assume that the adversary has found a vulnerability that eventually allows her to read from or write to an attacker-chosen address, the latter with data of her choice (i. e., an arbitrary read/write primitive). We further assume that the attacker operates in a scripting environment (such as JavaScript within a browser) and can interactively respond to events concerning the underlying application. In addition, we assume the attacker to have knowledge about the system configuration, including applied defenses, and the application's source code.

In addition to defensive mechanisms assumed in attacker models of previous works [10, 11, 30, 32], we assume a state-of-the-art exploitation mitigation such as Readactor++ to be in place:

– **Vtable Randomization and Booby Traps.** Vtables are randomized and their entries are hidden behind trampolines and interleaved with booby traps (cf. Section 2.3).
– **Writable $\oplus$ Executable Memory.** The system allows memory pages to be either executable or writable, but not both.
– **Execute-only Memory.** The system is able to mark pages as executable, but neither writable or readable. We expect the target application to mark code sections accordingly (e. g., trampolines).
– **Fine-grained ASLR.** The system provides capabilities to randomize applications at the function level (i. e., we assume a stronger randomization compared to standard coarse-grained ASLR).
– **Brute-forcing Mitigation.** We expect the application to actively respond to a detected attack, for example by preventing automatic restarts in response to hitting a booby trap.

3.2 Attack Overview

The attack makes use of the fact that most of the memory that is used for the dynamic loading mechanism has to be readable. Generally speaking, the attacker is able to re-implement the system's dynamic loading mechanics with the help of available scripting engine capabilities and therefore can obtain the address of critical functions like `system` in `libc` to execute a *function-reuse* attack. Harvesting code pointers is a critical step for most kinds of code-reuse attacks. For one, overwriting a code pointer might eventually enable control over the program counter, whereas proper combination of known pointers is a key ingredient of modern exploit chains. With the recent advance of defense techniques that limit the set of valid targets of an indirect control flow transfer, especially those code pointers become important which point to the *beginning* of a function. Given such pointers, an attacker can set up a *function-reuse* attack which reuses either whole functions or parts of it, starting from its entry up to a certain instruction [18,25,30]. To be able to re-implement the dynamic loading mechanics and resolve symbols of her choosing, the attacker has to obtain a module base address (ideally the base address of the main module). This can be done with the help of an existing information leak vulnerability or advanced offensive techniques like crash-resistant scanning primitives [14]. Given the module's base address, the attacker is able to parse the binary file format header information and traverse the structures that hold information about the loaded modules and the symbols they export. In case of the *glibc* runtime, the attacker is able to obtain the address of the `link_map` data structure [17], a linked list. From there, she can traverse the list to obtain the module base of any module loaded into the process (e. g., by comparing against its file name). Then, given the correct module base, she can resolve any exported symbol from the module by parsing the file format in memory with the help of the scripting environment. Since the loader requires these data structures to resolve symbols, they must be present in memory.

Summing up, our attack relies on the fact that symbol metadata, while required by the loader for functionality such as dynamic loading, poses a viable source of function pointer leaks which can be subsequently used when mounting a function-reuse attack.

3.3 Example: Bypassing Readactor++

As an example of how an attacker can abuse dynamic loading in a function-reuse attack against a state-of-the-art exploit-mitigation system, we show how to exploit a Readactor++-protected variant of the Chromium web browser in version 40.0. The software is running on Ubuntu 14.04 with Linux kernel 3.13 patched for EPT support. Note that we re-introduced the same bug (CVE-2014-3176) the original Readactor++ paper protected against [24].

The instantiation of our attack follows the basic steps laid out in Section 3.2. However, it has to account for some peculiarities of the exploit-mitigation system in use, Readactor++. The main challenges lie in the randomized layout of virtual function tables and the extensive use of execute-only memory. In summary, we are able to bypass trampolines which hide code pointers by manually resolving addresses of functions of interest. We use this facility to retrieve a function gadget used in a later part of our attack and a critical function used in our payload. Further, we are able to perform vtable *crafting* in order to bypass booby traps (which lie in randomized vtables). This is possible because Readactor++ does not restrict the set of vtables usable at a callsite. As the index into the vtable is randomized at the callsite, execute-probing most likely triggers a booby trap. Effectively, this yields no control over the entry that is called, preventing one from mounting vtable `reuse` attacks such as COOP [30]. Instead, we *craft* a *fake* vtable. Finally, we employ a new kind of crash resistance to probe the callsite's index and eventually execute our payload. This is implemented by abusing Chromium's process creation model.

Enabling Function-Reuse Attacks The general idea of our attack is to replace the *xvtable* pointer with a so-called *Entry-point gadget*, as introduced by Göktaş et al. [18]. The gadget (*EP gadget*, for short) designates a sequence of instructions starting at a function entry and spanning all instructions till the first indirect call or jump. To find a suitable gadget, we symbolically execute paths in exported functions that start at function entry and end at an indirect call site. Symbolic execution then lets us filter for paths where the indirect call target—and its parameters—depends on the first parameter of the analyzed function (passed via `rdi`). We found `_obstack_newchunk`, exported by `libc`, which eventually calls `[rdi + 0x38]` with `[rdi + 0x48]` as its first argument. If we can execute the gadget and enforce the following layout on the object in `rdi`, `system` gets called with the correct argument:

```
 byte ptr [rdi + 0x50] ← 1
qword ptr [rdi + 0x38] ← &system
qword ptr [rdi + 0x48] ← &system_argument
```

Our analysis is similar to the one used to discover functions for *function chaining* [14]. While our analysis yields several potential gadgets, we did not investigate their feasibility, as one function gadget is enough to successfully mount our attack. Because we target semantical properties only, the approach is invariant to fine-grained diversification as employed by Readactor++ and similar defenses.

Obtaining the Gadget We obtain read/write primitives by using an out-of-bounds access in `Array.concat()` and predict the allocation address of an object of type `XMLHttpRequest`. As we are equipped with an arbitrary read/write primitive, we can parse metadata of Chrome's allocator, PartitionAlloc, to calculate the bucket the next allocation of said object will be placed in.

The predicted object provides us with a pointer to an *xvtable*. It is important to note that these pointers point into Chromium's module range, so we can use them to deduce its base address. However, we anticipate that in the context of an attacker with crash resistance at her hand, it is hard to prevent leakage of module base addresses, as the attacker is able to scan the memory until a module base address is found. Since the memory is only read during the scan and not executed, active countermeasures like booby traps are ineffective.

Given the module's base address, we first obtain the pointer to the `link_map` structure, which contains information about all shared objects loaded into the process space. We can then walk the list manually to find the base address of the C standard library, `libc`, and resolve the EP gadget by walking the symbol table. Hence, we mimic the loader's functionality from the JavaScript context and manually perform all necessary steps.

Preparing the Object Figure 1 depicts the modifications we make to the predicted object. Essentially, we utilize *vtable crafting* to perform the attack. This is possible since *rvtable* pointers are not enforced to lie in specific memory pages or to belong to a set of whitelisted vtables. Hence, we can craft a *fake rvtable* on the heap and make its *xvtable* pointer point to the beginning of our EP gadget. Concretely, we add an offset $i \cdot (-5)$ to the EP gadget pointer such

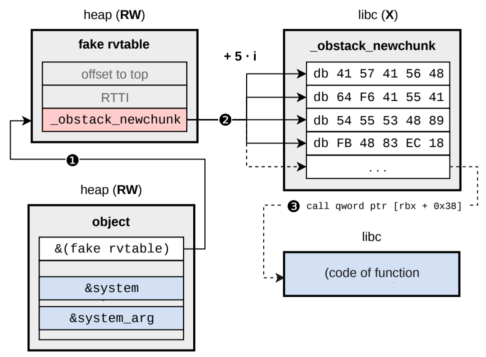

Fig. 1: Schematic overview of our attack: we perform a vtable crafting attack instead of a traditional vtable reuse attack. To this end, we put a fake *rvtable* in memory and modify our object to point to it (edge ❶). Also, we prepare the object according to the semantics of our EP gadget. In the *rvtable*, we replace the *xvtable* pointer to point to our EP gadget instead (`_obstack_newchunk`, edge ❷). If we hit a callsite to our object with an *applicable* index i, edge ❸ will call `system` with the given argument.

that the beginning of the gadget lies in some entry within the *xvtable*. Also, we modify the object according to the constraints given in Section 3.3. It is crucial to note that by *replacing* the *xvtable* pointer, we bypass booby traps completely. Booby traps in Readactor++ are interspersed among entries in the *xvtable*. Our object interprets the function `_obstack_newchunk` itself as *xvtable*. Therefore, callsites targeting the aforementioned object cannot trigger booby traps, hence this defense approach does not hinder the attack in any way.

Triggering the Gadget Having set up the object, triggering the gadget is not as straight-forward due to the randomized vtable layout in Readactor++. Depending on the callsite, the application will call our gadget with an offset of $5 \cdot i$, with $i \geq 0$ unknown (and limited by the number of entries in the *xvtable*). To keep the application alive despite faults due to the unknown vtable offset, we resort to a form of *inter-process crash resistance* by abusing the Zygote process creation model as used in Chromium.

On startup, Chromium forks a designated process, the so-called *Zygote*, to speed-up responsiveness when new tabs are opened [7]. In it, necessary shared objects are already loaded and initialized. If a new tab is opened, the Zygote forks a new process which is responsible for said tab (the *renderer process*). The forked process inherits the Zygote's memory layout and little additional initialization is required. We can use this behavior to our advantage and spawn the function-reuse attack outlined above. To this end, from one main tab, we spawn several tabs, each running our exploit for a varying value of i. If a tab crashes due to an access violation (i. e., our guess for i was wrong for this process), it does not influence any other tabs. Eventually, one tab will succeed and execute `system(·)`

via the EP gadget. Note that our attack is invariant to the fact whether i is randomized for the callsites in the renderer process after forking. We are not dependent on the exact same memory layout in each renderer.

Increasing Bruteforce Efficiency Crane et al. anticipated bruteforce attacks in the design of the Readactor system. Having to guess i correctly would mean the defender can scale the number of guesses an attacker has to perform linearly by adding more elements to the application's vtables. Considering the soft boundaries on memory usage nowadays, this is a reasonable way to increase the attack complexity a bit. Still, an attacker has multiple options. For one, she can simply choose the smallest vtable she is still able to trigger a virtual callsite for. But what is more, the correct choice of the EP gadget decreases the number of guessing attempts by a factor. In our case, `_obstack_newchunk` allows execution from offsets 0, +5, and +10 without impacting the semantics of calling the attacker's designated function. This effectively increases the chance of guessing one *applicable* index i by the factor of three.

4 Defense: Symbol Integrity

As discussed in Section 2.1, abandoning dynamic loading altogether is not an option in typical environments. Hence, we propose in the following an approach to restrict access to vital symbol information for every piece of code but the loader itself that is also compatible with legacy binaries. This effectively prevents an attacker from discovering any usable call targets using function exports and prevents her from setting up a usable payload.

A related defense is implemented in Microsoft's *Enhanced Mitigation Experience Toolkit* (EMET [23]). Its feature *Export Address Table Filtering Plus* (EAF+) restricts accesses to symbol information coming from blacklisted modules. However, several bypasses exist [2, 14, 27] which leverage functionality in white-listed modules to access export address tables. Further, since only specific pointers to the symbol table are protected by EMET, it is also possible to scan for symbol tables in memory directly. In practice, EAF+ might also lead to compatibility problems if a legacy module ends up on the blacklist for erroneous reasons. This motivates us to provide a more complete protection that does not rely on blacklists, but builds upon information hiding. In order to prove feasibility, we implemented our approach on top of the Readactor++ system in a tool called *Symtegrity*. Our defense successfully mitigates attacks relying on symbol resolution, such as the one presented in Section 3, and can be integrated into existing defenses given that it utilizes an orthogonal defense approach.

4.1 High-level Overview

The basic idea of Symtegrity is to leverage execute-only memory for hiding information about symbols. More specifically, we replace readable symbol metadata with references to so-called *oracles* that return the symbol address when executed. Since the oracles lie in execute-only memory, an attacker is not able to directly disclose them by reading the corresponding process memory (even in the presence of an arbitrary read/write primitive).

We implemented Symtegrity as a shared object for x86-64 Linux applications using the *glibc* standard library. Our implementation is compatible with legacy software and does not require source code access. Note that the general approach is not necessarily limited to Linux applications, but can also be implemented for different operating systems such as Microsoft Windows.

On startup, our defense processes each shared object needed by the protected application. In this step, the *relative virtual addresses* in symbol metadata (RVAs) are replaced by an index, whereas the RVA itself is stored in an execute-only mapping. Further, this mapping is updated on every shared object load. Entries in the mapping are not stored as data, but code: each entry is an *oracle* that, when executed, yields the data point as return value. The defense also hooks into various functions in the loader which should yield the RVA of the function it has been queried for (the full list is given in our implementation). In each of these functions, the hook re-translates the index back to the original RVA at runtime by querying the corresponding oracle. Thus, at no point during execution, the original symbol RVA is available to the attacker. It is only passed via ephemeral local variables inside those functions in the loader that are responsible for resolving symbols, such as `dlsym`. If an attacker already gathered the address of this function, she could resolve symbols legitimately. Our defense cannot protect applications that, by themselves, give access to such symbol resolution oracles. In the following, we briefly discuss several implementation details.

Hiding the Mapping The mapping is allocated at program startup, or, in case of forking applications, at the startup of the child process. This way, we can ensure that we make proper use of ASLR as provided by the OS. Also, we mitigate shortcomings of ASLR implementations in presence of the Zygote model by re-randomizing the child process [20]. Further, the first entry in the mapping does not start directly at the mapping base, but is shifted down by a randomly chosen offset Δ (cf. Figure 2). Indices are relative to the shifted mapping and do not leak the range of Δ. Even if an attacker was able to deduce the mapping base (i. e., the first page), she would not be able to call the oracle corresponding to an index she retrieved from the symbol table. Finally, in spirit of Readactor++, the mapping is guarded by inserting booby traps in between benign oracles. This prevents an attacker from randomly querying oracles in a page she assumes to be the mapping page and also prevents linear scans via read/write primitives and blind/crash-resistant execution/probing. Effectively, to probe any index, she would previously have to guess Δ in order to obtain the address of the first oracle. All other oracle indices are relative to that one. However, since she has to resort to execution in order to probe entries, she will eventually hit the booby trap and the attack is successfully detected. Note that this is different from our assumption that an attacker is able to figure out the base address of *modules* of her choice: in this case, she merely reads at particular addresses in a crash-resistant manner. In case of the mapping, however, she has to *execute* code while being able to recover from crashes. Due to the presence of booby traps this is an adversarial scenario, since they actively react to probing attempts. Counteracting this requires a much stronger crash resistance primitive compared to the case of module base discovery.

Updating the Mapping Upon startup, the defense processes all shared objects which are needed by the application and updates the mapping accordingly. The same happens at runtime once a new module is loaded. Each symbol gets assigned a random, unique index into the mapping (seeded anew for each process). The index indirectly refers to an oracle returning the RVA of the symbol. This is achieved by updating the `ElfW(Sym)` structure, which stores all metadata related to a symbol. Its `st_value` field contains the relative address from the module's base address to the symbol itself. Thus, the defense writes the assigned index into the symbol's `st_value` field and assembles an oracle of the following form in the mapping: `mov rax, __real_rva; ret`.

Note that during an update, the affected mapping page is unprotected for a small time window. However, we aim to keep that window as small as possible. Concrete empirical measurement results are presented in Section 4.2. The results show that the window is far too small to be of practical use for an attacker and we provide a more extensive discussion in Section 5.2. To mitigate an attacker deliberately reloading shared objects to increase her chances, one could cache the mapping per object and re-use it. As the mapping remains execute-only, the attacker cannot prolong the update window. However, due to caching, indices would not be re-randomized on the next library load. This does not help the attacker though, as she has no knowledge about the mapping itself.

Symbol Translation Figure 2 depicts the translation process. While multiple loader functions are hooked, we will describe the process using the example of `dlsym`; other hooks are implemented in the same way. The hook operates at the epilogue of `dlsym`. Hence, it obtains both the proper base address the requested symbol lies in as well as the "encoded" index (due to our modifications, `dlsym` returns a seemingly valid address of the form `base + index`, from which we can deduce the index). From the index, Symtegrity calculates the offset into the mapping by multiplying it with the size of an oracle. The oracle is then executed and yields the real RVA in `rax`. The hook now adds the object's base address to `rax` and returns the resulting value, which is the full address to the symbol.

This sort of translation is applied to all relevant points in the loader and supports all dynamic linking and loading features we encountered in common applications, such as lazy binding, which resolves symbols on demand.

4.2 Evaluation Results

We evaluate the performance of Symtegrity on 64-bit Chromium 40.0 protected with Readactor++. It runs on Ubuntu 14.04 with Linux kernel version 3.13 with

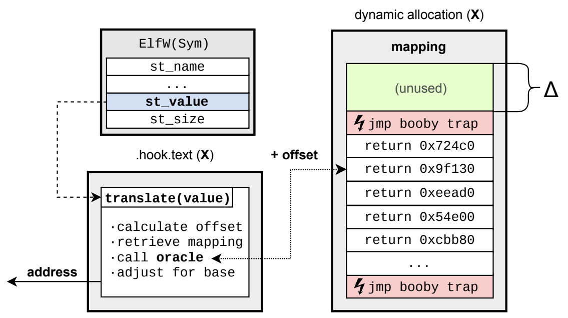

Fig. 2: The `translate` function receives an address encoding the index, with which the defense previously overwrote the `st_value` field. The function proceeds by scaling this index and use it as an offset into the mapping. By simply executing the function at said offset, it receives the original RVA. After adding the corresponding base address to it, the address is returned.

the Readactor patch. Due to the hypervisor used in Readactor++, we only use one core of our Core i7 CPU clocked at 1.2 GHz and disabled advanced features such as hyper-threading, an implementation requirement of Readactor++. 8 GiB of physical memory are available on this test machine.

Startup Time In order to measure the baseline for the startup time, we added a dependency on a custom shared object to Chromium and ensured that this benchmarking library is initialized first. This enables us to measure the exact time throughout all stages of the Zygote process creation model. The library measures the path from the startup of Chromium to the entry point in the renderer process spawned for the initial tab. This includes the time to fork and initialize the Zygote, the Zygote to fork into a new renderer process, and to execute till the entry point of the renderer process. In a second run, we repeat the measurements with our defense applied. Finally, we take the geometric mean over 100 runs.

Overall, Symtegrity induces an overhead of roughly 8% at startup. The unprotected Chromium takes about 6.9 seconds to finish this measurement, whereas the protected variant takes about 7.5 seconds on average. Note that this slowdown mostly impacts the startup time of the Zygote, which is done only once when starting the browser. Furthermore, the startup overhead is dependent on the number of dynamically linked shared objects. An application that does not link as many shared objects is subject to a lower overhead on startup. When the Zygote process is already running, the additional delay in opening a new tab with our defense in place is about 0.3 seconds. This delay is hardly noticeable by a user working with the browser.

Load Time To further quantify the impact of our defense, we measured the average time Symtegrity takes to update the symbol table of a newly loaded shared object. Hence, we micro-benchmarked the function that processes every object at Chromium startup (i. e., the startup of the Zygote process). Note that the timings are also representative for the workload performed if an object is loaded dynamically via `dlopen`. On startup, Chromium loads 84 shared objects of varying size, which are all linked directly to the application. To set things into perspective, the objects export 51,873 symbols which all have to be processed by Symtegrity. Measuring the processing time for all 84 shared objects loaded on startup yields a geometric mean of roughly 743 µs per shared object. Given that new shared objects are not loaded as often at runtime as during the application's startup phase, we deem the performance overhead reasonable.

Absolute Calls We counted the number of actual calls dispatched to functions related to symbol resolution. During startup, 7 calls to `dlopen`, 43 calls to `dlsym`, and 217 calls to `_dl_fixup` were recorded. The first two APIs are manifestations of *dynamic loading*, as discussed in Section 2.1. We emphasize that these calls do not include shared objects loaded due to *dynamic linking*. The latter API is called when a lazily-bound symbol is requested for the first time.

Mapping Update Finally, we measured the time in which parts of the mapping are accessible via memory reads. This situation occurs every time the mapping is updated (i. e., a new module is loaded for which oracles are inserted into the mapping). We measured the time from which on write permissions are in place (which imply read access as well) up to the point where permissions are switched back to execute-only. We noticed differences when measuring the window and hence distinguish two phases: one from application startup till the start of the renderer process and another one from that point onwards. For each phase, we averaged the results across multiple runs. For the first phase, covering the application's startup, 468,687 update events have been recorded. The geometric mean of the time window in which the mapping is unprotected is around 1,623 ns, whereas the average value is around 4,149 ns. Consequently, we detected 316 outliers that diverge by more than one standard deviation, where the maximum value lies at 9.948 ms. The second phase covers the time frame from the renderer's entry point to the point where it successfully loaded a page. Naturally, we recorded fewer events. The geometric mean across 7,200 data points is around 2,104 ns, the average value around 2,123 ns. As both timings are close to each other, fewer outliers were found. The maximum value of all 83 outliers lies at 0.0605 ms. While we are unable to give concrete evidence as for the root cause of the higher number of outliers during startup, we suspect them to be due to the high amount of mapping updates and a result of scheduling events. Still, as an attacker does not yet have control over the application during startup, we do not deem this a shortcoming of our approach.

Evidently, the duration in which the mapping is unprotected during an update is far smaller after startup. We argue that this time window is far too small to be of practical use for an attacker. A detailed discussion about the feasibility of this attack and countermeasures is given in Section 5.2.

Benchmarks To evaluate Symtegrity's performance overhead, we ran a SPEC CPU2006 INT benchmark, using SPEC version 1.1. Specifically, we averaged over 10 runs using the *ref* test set and 3 iterations per individual benchmark. The top of Table 1 shows the results (lower numbers are better). It shows the runtime of runs using both the unprotected program (second column) and the one protected by Symtegrity (third column) as well as the relative overhead. The overall overhead is based on the geometric mean of the individual benchmark results. As evident from the table, our prototype implementation incurs almost no observable overhead. This is expected, as the SPEC benchmarks are used to measure performance overhead for computationally intensive tasks. Symtegrity, however, only impacts runtime negatively during startup and upon calls to dynamic loading facilities.

Table 1: Results of the SPEC CPU2006 INT benchmarks (top; lower numbers are better) and JetStream (bottom; higher numbers are better).

SPEC Benchmark	Runtime (s)	+Symtegrity (s)	**Rel. Overhead**
400.perlbench	259	258	−0.55%
401.bzip2	386	387	+0.22%
403.gcc	246	247	+0.37%
429.mcf	289	288	−0.27%
445.gobmk	390	391	+0.36%
456.hmmer	367	368	+0.38%
458.sjeng	422	422	−0.18%
462.libquantum	316	316	+0.07%
464.h264ref	433	435	+0.42%
471.omnetpp	321	322	+0.39%
473.astar	345	347	+0.40%
483.xalancbmk	203	201	−0.80%
Overall SPEC (geometric mean)	323.62	323.84	+0.0649%
Latency	43.014 ± 3.2720	42.641 ± 0.7330	+0.875%
Throughput	134.82 ± 3.9437	135.36 ± 1.5001	−0.399%
Overall JetStr.	81.934 ± 3.9436	81.812 ± 0.8489	+0.149%

We also conducted a benchmark using the JetStream 1.1 JavaScript benchmark suite [34]. It combines several well-known benchmarks, such as SunSpider, Octane, and those of LLVM, and yields scores for latency, throughput, and an overall score [34]. The bottom of Table 1 shows the results (higher numbers are better). To perform the benchmark, we started the Chromium browser protected by Readactor++ and ran the JetStream benchmark suite to obtain the *Base* column. We repeated the same process using the Readactor++-protected browser with Symtegrity on top to obtain the measurement values in the column titled

+*Symtegrity*. JetStream runs each individual test three times and reports a score for each, along with the specific scores shown in Table 1. Evidently, the overall score of our defense lies well within the uncertainty of the base score, indicating that no measurable overhead is introduced when considering the performance of JavaScript in the browser. These results are expected, as Symtegrity mainly affects the browser's startup time.

To quantify the amount of new libraries that need to be hooked by our defense when rendering a web page, we visited the global top 500 pages, as reported by Alexa [1]. We allocated a time span of six seconds for each individual site to load. All in all, only one new shared object, `libfreebl3.so`, would be loaded dynamically, for 9 out of 500 pages. This is also due to the fact that Chromium aims to optimize load times by pre-emptively loading objects on start up, as seen in Section 4.2. This supports the fact that at runtime, only few (comparatively costly) updates to the mapping have to be performed by Symtegrity.

5 Discussion

5.1 Scope and Limitations of our Defense

Symtegrity focuses on restricting access to vital symbol metadata to the respective functions in the loader. It is developed as an orthogonal defense mechanism that is a crucial building block for state-of-the-art defenses that still expose aforementioned data to an attacker. Our approach assumes that the protected application does not expose an oracle to the attacker which is capable of leaking symbol addresses. This covers both intentional oracles, which can be deemed unlikely to occur in common applications, as well as any kind of side-channel. For example, we suggest using *eager binding* in order to prevent exposing an address oracle via resolving functions, such as proposed by Readactor++. Still, detecting such cases in an automatic fashion arguably is hard and thus cannot be covered by Symtegrity. This poses the greatest limitation of our approach.

Furthermore, in a full exploit-mitigation system, one also has to prevent the attacker from disclosing function addresses directly. Since Symtegrity focuses on preventing an adversary from resolving the symbol addresses manually by using the available metadata, attacks using function addresses disclosed previously are out of scope. Hence, Symtegrity is an important piece of the defense puzzle rather than a full-fledged protection system on its own. To this end, we tested an exemplary configuration in which we applied Symtegrity to a Readactor++-protected application. Note that an attacker is unable to use any function pointer in the Procedure Linkage Table (PLT) as it is randomized and interspersed with booby trap entries. Therefore, in the case of Readactor++, she cannot use symbol-resolving functions such as `dlsym` even if the application intentionally exposes them. In the presence of such a defense, more sophisticated attacks that directly scan the remaining readable memory to deduce symbol addresses from its metadata are mitigated.

In the current design, an attacker can potentially correlate the oracle index (found in the `st_value` field) with the symbol itself, e. g., via the `st_name` field of the `ElfW(Sym)` structure. However, it is important to note that possible attack scenarios using this correlation make assumptions that are difficult to satisfy in practice. Assuming that the attacker knows both the symbol mapping's base address as well as the offset $\varDelta$ into the mapping (at which the first oracle lies), she can query any oracle to retrieve the original RVA. Consequently, in its current state, Symtegrity's security relies on keeping the base address and $\varDelta$ secret. This is achieved by the underlying system's memory layout randomization capabilities, execute-only memory, as well as booby traps guarding the mapping against execute-probing. Still, the attacker's ability to correlate symbols and oracle indices can be counteracted by extending the current prototype system with two features. First, we can protect the `st_name` field in the same manner as we already do with the `st_value` field. A second extension would involve randomization of the layout of the symbol table in memory. This prevents attacks where an attacker assumes the *i*-th function exported by, e. g., `libc`, to be `system`. By applying the same reasoning to the *non-randomized* symbol table in memory, she would still be able to correlate function and oracle index for a specific version of the library, despite the protections in place. However, we assume the current state of protection (randomization of base addresses, execute-only memory, random offset $\varDelta$, and booby traps) to be sufficient. Given these assumptions, an attacker is not able to query the oracle corresponding to a symbol of interest in the current approach. As the aforementioned extensions would impact the system overhead negatively, we decided not to include them at this point.

5.2 Data Race on Mapping Update

During dynamic loading, Symtegrity has to change the access permissions of the mapping containing the symbol address oracles. This leaves them unprotected for a small time frame in which an attacker could leak information by direct memory disclosure (see Section 4.2). Namely, she would obtain the real symbol RVA returned by an oracle. However, the attacker has to overcome multiple hurdles before being able to do so: First, she must be able to trigger the process in which the mapping gets updated in the first place, i. e., load a new module into the process. This is not a common feature in applications that can be triggered arbitrarily by an attacker. In our tests only one shared object was dynamically loaded after the startup process (see Section 4.2). Furthermore, even if the attacker can trigger the dynamic loading of a shared object, she has to be able to do it multiple times in order to win the race. Second, she has to know the base address of the mapping and the random offset $\varDelta$ upfront. Third, the disclosure has to be fast enough to win the race against the defense re-protecting the affected page with execute-only permissions. We consider these conditions unlikely in practice.

5.3 Applicability of our Attack

In Section 3.3, we presented an exemplary attack on the state-of-the-art exploit mitigation system Readactor++. We argue that our attack is feasible against other defenses and hence we now discuss the applicability of our attack to related approaches which also leverage information hiding for exploit mitigation.

HideM and XnR Gionta et al. presented *HideM*, a system implementing information hiding in order to mitigate memory disclosure vulnerabilities [15]. This is achieved by leveraging the TLB split mechanism which allows HideM to serve different views into a page, depending on the type of access (data read or instruction fetch). Similarly, Backes et al. presented *XnR* (*Execute-no-Read*) [3], which, to some extent, achieves a similar form of information hiding based on *non-present* pages and a page fault handler. Both approaches aim to mitigate direct disclosure of executable pages by either returning dummy values upon a read or preventing read access altogether. Consequently, our attack is directly applicable to either of the aforementioned systems. At no point during the attack, direct disclosure of code pages is required.

Heisenbyte Tang et al. presented *Heisenbyte*, along with the concept of *destructive code reads* [33]. The main idea is that upon a read, the operation is served, but the corresponding bytes are replaced by random values. Effectively, this ensures that reads serviced from executable pages do no longer correspond to the bytes that are executed at the very same address. What distinguishes this system from *HideM* and *XnR*, regarding our attack, is the fact that it also assumes load-time fine-grained ASLR. Our attack, however, neither reads executable pages nor does it make assumptions about the memory layout other than the address at which an exported function (such as `_obstack_newchunk`) lies. Hence, it should be applicable to Heisenbyte as well. Still, we recognize this as an underlying requirement for all fine-grained randomization-based systems: Such approaches have to be aware of any references into the code in order to keep the program intact. A concrete instantiation would be support for dynamic loading and symbol resolution. The aforementioned issue requires such systems to either refrain from randomizing referenced locations such as function start addresses or to deliberately update the references to be synchronous with the new memory layout.

TASR Bigelow et al. presented *TASR*, a system to re-randomize an application's memory layout at predefined points in time [4]. TASR observes a set of system calls associated with either category and re-randomizes the memory layout every time one of these syscalls is requested. Making a definite statement about the applicability of our attack in presence of TASR is difficult. For one, TASR does not explicitly take JIT engines into account. Consequently, applicability highly depends on where the I/O boundary is placed between the scripting engine and the native context. More specifically, searching the module base would most likely not trigger re-randomization as no syscalls are involved. Still, triggering the object's callsite may very well involve syscalls along the way and re-randomize the memory layout. Also, the validity of obtained symbol RVAs may be affected, depending on the way such references are kept synchronous. In the end, one has to consider trade-offs in order to meet the performance requirements imposed upon the scripting facilities in modern browsers when placing the I/O boundary. Unfortunately, TASR's implementation is not available such that we have to discuss these aspects solely on the claims made in the original paper.

5.4 Limitations of our Attack

While the attacks presented in Section 3.3 primarily serve as a proof-of-concept to demonstrate how an attacker can abuse dynamic loading, we discuss in the following some practical hurdles we came across in the Readactor++ bypass (see also our TR [8]). First, we had to disable Chromium's popup blocker. With the popup blocker in place, our main tab would not have been able to spawn the additional tabs that effectively brute-force the unknown index i. While this is a limitation of our attack against the Readactor++-protected Chromium, it is no inherent limitation of the underlying attack vector. However, spawning several tabs of which one is the succeeding exploit and others abort abnormally is usually prevented, because browsers do not restart on crashes. This was solved with inter-process crash resistance, and could be prevented by locking the system after exceeding a specific number of tab crashes. However, browser vendors trade off usability in favor of security in this case. Note that we only chained together two exported functions: `_obstack_newchunk` is used as first gadget to execute `system` at a C-style callsite, to perform our attack. Chaining multiple EP gadgets does not impact bruteforcing, i. e., we do not need to spawn more tabs. This is due to the fact that we can gather exports (code pointers) with *read* instructions (mimicking `dlsym`) from JavaScript. We only need to bruteforce the randomized index in the fake vtable to *start* the chain (i. e., *execute* a fake virtual function, our first gadget). This is done to circumvent booby traps. The subsequent EP gadgets are called at C-style callsites in the current gadget and are unprotected in *Readactor++*. However, if an attacker wants to chain EP gadgets which call subsequent gadgets at virtual function callsites, bruteforcing attempts would increase drastically, because different virtual function callsites would need to be bruteforced for the appropriate virtual function index. This is likely to reduce exploit success. Hence, we did not pursue such gadget chains further. For our attack, we disabled Chromium's `seccomp` sandbox. Obviously, it limits our ability to dispatch specific system calls, but does not prevent code execution per se, which was the sole aim of our function-reuse attack.

6 Related Work

We already discussed several papers closely related to our work and review several other related papers in the following. Similar to Readactor, execute-only memory was implemented for mobile devices by LR^2 [5]. By preventing attackers to use load instructions and hiding pointers to code, protected programs become resilient against memory disclosures. While this defense introduces execute-only memory to a different architecture, our approach is complementary as it enables protection of those functions that have to remain at discoverable locations, i. e., exports. Similarily, kRˆX introduces execute-only memory to protect against code-reuse in the kernel [28], while our approach aims to protect against adversaries targeting complex userspace programs such as browsers. *Shuffler*, a re-randomization scheme thwarting JIT-ROP, is complementary to our defense, since we target an adversary trying to gather export symbols. It continuously changes code locations, and hence, impedes code-reuse attacks significantly [35]. Due to the time delay between memory disclosures used to build the attacker's gadget chain and its execution, the payload will fail, as the gadgets are elsewhere in the address space. A binary-only code diversifier, dubbed *CodeArmor*, protects code pointers against disclosure and re-randomizes code mappings to hinder code-reuse attacks [6]. Our defense is different as it aims at protecting symbol mappings, without significant run-time overhead. While more narrow in scope, it complements existing (compiler-based) code pointer hiding techniques.

ASLR-Guard [21] also uses information hiding to protect code locations. In contrast to Readactor, it does not require execute-only memory. Their approach assumes fine-grained ASLR and encrypts all code pointers to prevent an attacker from leaking information. The concept of code pointers encryption [22] can alternatively be used for Symtegrity when execute-only memory is not available. Instead of replacing symbol metadata with oracles in execute-only memory, one could encrypt the metadata with a key *hidden* in memory, whose location is only known to the symbol resolving functions. Unfortunately, this approach is vulnerable to adversaries which have a strong memory scanning primitive such as crash resistance at hand.

The attacks proposed by Di Federico et al. [12] are similar in spirit to the one we propose, as they also leverage the *glibc* runtime's symbol resolving capabilities. Instead of manually walking through the symbol resolution process, they use the internal functions responsible for resolving the addresses in the first place and call them on manipulated metadata. However, our attacker model does not cover the case where the corresponding function addresses already leaked to an attacker. Furthermore, their approach assumes that the targeted application is not a position-independent executable (PIE) and lazy binding is used either by the target binary itself or by at least one of the shared objects it depends on. This prerequisite is not required by our approach. Rudd et al. [29] recently propose a novel class of code-reuse attacks. Similar to our work, feasibility of their attack is shown on the example of Readactor and, in particular, indirect code pointers as implemented using, e. g., trampolines. However, instead of brute-forcing the randomized index into a vtable, they *profile* trampoline pointers and are subsequently able to induce malicious behavior using indirect code pointers.

Another defense that targets the loader itself is *Safe Loading* by Payer et al. [26]. They replace the default loader with a hardened one which effectively acts as a user-space sandbox. Modifications include restriction of existing functionality, such as preloading, and the addition of new security-related functionality, such as indirect branch checking. Nevertheless, the symbol resolution process still makes use of a module's symbol table. Consequently, our approach can be used to further refine the safe loader.

7 Conclusion

In this paper, we showed how existing mitigations can still be bypassed by proposing a novel attack vector based on symbol metadata. To demonstrate that such attacks are indeed feasible in practice, we provide a concrete instantiation of our attack against Readactor++ and are the first to demonstrate how the general concept of this defense can be bypassed. Furthermore, we mitigate the attack vector by replacing easily-accessible metadata with oracles located in execute-only memory that yield the corresponding value upon execution. Future defenses need to take symbol metadata into account given that this attack vector turns out to be the Achilles' heel of state-of-the-art defenses.

References

1. Alexa Internet, Inc. Top 500 Sites on the Web. http://www.alexa.com/topsites.
2. A. Alsaheel and R. Pande. Using EMET to Disable EMET. https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disabl.html.
3. M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run But You Can't Read: Preventing Disclosure Exploits in Executable Code. In *ACM CCS*, 2014.
4. D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *ACM CCS*, 2015.
5. K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen. Leakage-Resilient Layout Randomization for Mobile Devices. In *NDSS*, 2016.
6. X. Chen, H. Bos, and C. Giuffrida. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *IEEE EuroS&P*, 2017.
7. Chromium. Usage of the Zygote Process Creation Model in Chromium. https://chromium.googlesource.com/chromium/src/+/master/docs/linux_zygote.md.
8. M. Contag, R. Gawlik, A. Pawlowski, and T. Holz. Technical Report: On the Weaknesses of Function Table Randomization. Technical report, Ruhr-Universität Bochum, 2018.
9. S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby Trapping Software. In *ACM Workshop on New Security Paradigms (NSPW)*, 2013.
10. S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE S&P*, 2015.
11. S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRAP: Table Randomization and Protection against Function Reuse Attacks. In *ACM CCS*, 2015.
12. A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. How the ELF ruined Christmas. In *USENIX Security*, 2015.
13. I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *IEEE S&P*, 2015.

14. R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*, 2016.
15. J. Gionta, W. Enck, and P. Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *ACM CODASPY*, 2015.
16. C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization. In *USENIX Security*, 2012.
17. glibc. link.h header file, defining `link_map`. `https://github.com/bminor/glibc/blob/master/include/link.h`.
18. E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *IEEE S&P*, 2014.
19. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *USENIX OSDI*, 2014.
20. B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *IEEE S&P*, 2014.
21. K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *ACM CCS*, 2015.
22. A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *ACM CCS*, 2015.
23. Microsoft. The Enhanced Mitigation Experience Toolkit. `https://support.microsoft.com/en-us/kb/2458544`.
24. National Vulnerability Database. Vulnerability Summary for CVE-2014-3176. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3176`.
25. Nergal. The advanced return-into-lib(c) exploits: PaX case study. `http://phrack.org/issues/58/4.html`.
26. M. Payer, T. Hartmann, and T. R. Gross. Safe Loading – A Foundation for Secure Execution of Untrusted Programs. In *IEEE S&P*, 2012.
27. Piotr Bania. Bypassing EMET Export Address Table Access Filtering feature. `http://piotrbania.com/all/articles/anti_emet_eaf.txt`.
28. M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis. kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *ACM European Conference on Computer Systems (EuroSys)*, 2017.
29. R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, et al. Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity. In *NDSS*, 2016.
30. F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE S&P*, 2015.
31. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *ACM CCS*, 2004.
32. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *IEEE S&P*, 2013.
33. A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *ACM CCS*, 2015.
34. WebKit. JetStream JavaScript benchmark suite. `http://browserbench.org/JetStream/`.
35. D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *USENIX OSDI*, 2016.