Dynamic Loader Oriented Programming on Linux

Julian Kirsch, Bruno Bierbaumer, Thomas Kittel
Technical University of Munich
Garching, Germany

Claudia Eckert
Fraunhofer AISEC
Garching, Germany

ABSTRACT

Memory corruptions are still the most prominent venue to attack otherwise secure programs. In order to make exploitation of software bugs more difficult, defenders introduced a vast number of post corruption security mitigations, such as w⊕x memory, Stack Canaries, and *Address Space Layout Randomization* (ASLR), to only name a few. In the following, we describe the *Wiedergänger*[1]-Attack, a new attack vector that reliably allows to escalate unbounded array access vulnerabilities occurring in specifically allocated memory regions to full code execution on programs running on `i386`/`x86_64` Linux.

Wiedergänger-attacks abuse determinism in Linux ASLR implementation combined with the fact that (even with protection mechanisms such as *relro* and *glibc*'s pointer mangling enabled) there exist easy-to-hijack, writable (function) pointers in application memory. To discover such pointers, we use taint analysis and backwards slicing at the binary level and calculate an over-approximation of vulnerable instruction sequences.

To show the relevance of Wiedergänger, we exploit one of the discovered instruction sequences to perform an attack on Debian 10 (*Buster*) by overwriting structures used by the dynamic loader (*dl*) that are present in *any* application with *glibc* and the dynamic loader as dependency. In order to show generality, we solely focus on data structures dispatched at program shutdown, as this is a point that arguably all applications eventually have to reach. This results in a reliable compromise that effectively bypasses all protection mechanisms deployed on `x86_64`/`i386` Linux to date.

We believe Wiedergänger to be part of an under-researched type of control flow hijacking attacks targeting internal control structures of the dynamic loader for which we propose to use the terminology *Loader Oriented Programming* (LOP).

KEYWORDS

Software Exploitation, Software Security, Software Vulnerability, Loader Oriented Programming, Dynamic Loader, Address Space Layout Randomization Determinism, glibc, Linux

[1] ˈviːdəˌgɛŋə (lit. *"One Who Walks Again"*)

ROOTS, November 16–17, 2017, Vienna, Austria

ACM ISBN 978-1-4503-5321-2/17/11…$15.00
https://doi.org/10.1145/3150376.3150381

ACM Reference Format:
Julian Kirsch, Bruno Bierbaumer, Thomas Kittel and Claudia Eckert. 2017. Dynamic Loader Oriented Programming on Linux. In *Proceedings of Reversing and Offensive-oriented Trends Symposium, Vienna, Austria, November 16–17, 2017 (ROOTS),* 13 pages.
https://doi.org/10.1145/3150376.3150381

1 INTRODUCTION

In current software projects that use the C programming language callback mechanisms are frequently used to execute functionalities once a certain event occurs. For example the `atexit` function as described by the *Portable Operating System Interface* (POSIX) family of standards allows programmers to register a function at runtime that will eventually be called upon program termination. Such functionality is typically implemented by means of function pointers that get dispatched by the C runtime once the program exits. In case of such *hooks* being stored in writable memory, an attacker is able to change the control flow of an application if they are able to overwrite the hook with a malicious value. In the worst case, an adversary can escalate a single overwritten hook to arbitrary code execution on the attacked system.

The above attack is a well-known technique when it comes to exploiting bugs in software. Unsurprisingly, a lot of effort has been spent to protect these hooks: For instance, they are typically stored in memory regions that do not contain data that is directly modifiable by the user. Furthermore, ASLR has been introduced to randomize and thus hide the absolute addresses of data in memory. Last, function pointers themselves are typically protected by the C runtime by either marking the memory they reside in as *read-only* or by using protections that scramble the pointer values using a secret key. However, as we will discuss in this work, even if all defenses are in effect, there *still* exist hooks that can be attacked.

This study focuses on *Unbound Array Access Vulnerabilities* — programming errors which lead to an array being accessed at an index outside of the range $[0, n-1]$ with n being the length of the array[2]. More specifically, we develop attacks on (erroneous) software such as

```
unsigned char *ptr = malloc(0x200000);

size_t idx = 0; unsigned char val = 0;
scanf("%zu␣%hhu", &idx, &val);

ptr[idx] = val;
```

where a malicous attacker can (repeatedly) control the index `idx` used to write (byte) value `val` into an array pointed to by `ptr`.

[2] Note how this is different from classic *Write-Anything-Anywhere*-bugs where an attacker can choose absolute addresses to write to.

The contributions of this work can be summarized as follows:

- We show that current userspace software make use of a multitude of different hooks during normal execution. In order to systematically detect hooks being dispatched, we propose a technique to extract all callback pointers of a program that are stored in writable memory based on backwards taint analysis and backwards program slicing.
- We also show that due to a lack of randomness of the ASLR implementation currently used by Linux, it is possible to exploit ASLR determinism in such a way that calculating the locations of interesting callback pointers a priori is feasible.
- Finally, we present Wiedergänger, a proof-of-concept implementation of an attack that targets writable pointers used by the C standard runtime environment (*glibc* and the dynamic loader) on current Linux systems even in presence of protection mechanisms built into current operating systems and compilers. The general idea of Wiedergänger is to corrupt code pointers that are later used by the runtime environment during application shutdown, and are therefore *generic to the whole software ecosystem.*
  We describe two examples of Wiedergänger-attacks allowing to spawn a shell on any *glibc*-based application that exhibits a *Write-Anything-Relative-To-A-Base-Address*-primitive (i.e. an unbounded array access vulnerability). Our first example has a worst case success rates of $\frac{1}{4096}$ while the second example always succeeds.

2 BACKGROUND

In this section, we briefly highlight the concepts needed to understand the scenario in which a Wiedergänger-attack can be conducted.

2.1 Exploit Mitigations

In current software systems, several mitigation strategies that aim to reduce the impact of potentially abusable programming errors are in use. The following concepts are important for our work:

2.1.1 w⊕x Memory. The idea behind w⊕x is that no memory within the software system should be *writable* and *executable* at the same time. The motivation is to deny an attacker the ability of first injecting arbitrary code into the process memory followed by executing this so-called *shellcode* [2] afterwards. On Intel `x86`, w⊕x is implemented by means of the *execute-disable* (`XD`) bit, which allows operating systems to forbid instruction fetches from particular pages.

2.1.2 Address Space Layout Randomization. In order to reduce the attacker's knowledge of interesting targets within a particular process, current operating systems randomize the location of stack, heap and libraries in memory at a per-execution basis. Some implementations of ASLR additionally randomize the image base address of the executable in memory, usually referred to as *Position Independent Executable* (PIE) binaries. It is important to note that on Linux (and on many other operating systems) ASLR operates at the granularity of virtual memory *pages*. This has the consequence that ASLR only randomizes those bits of a virtual address that are located beyond the position corresponding to the page size.

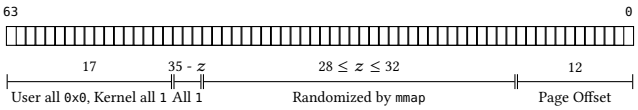

Figure 1: Address bits randomized by the `mmap` system call on `x86_64` Linux. Earlier kernels (≤ `4.5`) hard-code z to 28, newer kernels can be configured via the `/proc/sys/vm/mmap_rnd_bits` runtime parameter.

For example, as the size of a page of virtual memory used by Linux running on the `x86` architecture is characteristically $4096_d = 2^{12}$ Bytes, ASLR is only capable of randomizing bits beyond the bit at position 11 (counting zero-based). At the time of writing, current `x86_64` processors support only 48 out of 64 possible bits of virtual address space [5], with Linux setting the topmost (47th) bit (and therefore all bits beyond this position, due to canonicalization) to 1 for kernel and to 0 for user space addresses. Out of the $48 - 12 - 1 = 35$ remaining address bits, older `x86_64` Linux kernels (before `4.5`) randomize 28 bits[3], whereas newer kernels (starting with `4.5`) provide a run-time parameter[4] offering the possibility to increase this number to 32 bits. In practice, this raises the bar for brute-force guessing of addresses to a level that is considered sufficiently high in practice. With Intel's plans to increase the physical address width to 57 bits [3], ASLR can be expected to increase in strength in the near future. Figure 1 shows the randomized address bits returned by the `mmap` syscall when allocating new pages for a process.

2.1.3 Function Pointer Protection. As function pointers pose a promising target to gain control of the execution flow, it is desirable to deny attackers the ability of compromising them. To implement this, two mechanisms are currently in place:

The first, *Pointer Encryption* [6], introduces a per-process 64 bit random secret that is used to mangle pointers in memory. The mangling transformation is chosen such that the real pointer value can be derived from the mangled value in memory and the secret value, placing the result in a register. This *de-mangled* value is then used as target for an indirect control transfer. Note that Pointer Encryption is implemented in an ad-hoc manner: It is the responsibility of *the programmer* to perform the mangling. In practice, *glibc* mangles writable function pointers residing in global static memory (`bss`).

The second way of protecting function pointers is to place them in memory marked as read-only at runtime (note that this is different from loadtime). This mechanism, on Linux typically referred to as *relro* (relocations read-only), enforces global static function pointers contained in the *Global Offset Table* (GOT) and *Destructor* (DTOR) section of a binary to be mapped read-only. Note that this mechanism forces lazily operating dynamic linking systems (such as the dynamic loader on Linux) to resolve any relocations to external functions at program startup.

[3] `arch_mmap_rnd` in `arch/x86/mm/mmap.c`
[4] `cat /proc/sys/vm/mmap_rnd_bits`

2.2 The Arms Race in Abusing Software Vulnerabilities

When exploiting vulnerable software systems, an attacker's goal is usually to take control of the program's execution flow. The reasons that this becomes possible are manifold: Classic stack-based buffer overflows can lead to exploitable conditions, as can format string vulnerabilities, or the corruption of function pointers in memory, just to name a few. However, during most breaches, attack methodologies converge to a point where an attacker can arbitrarily control the contents of the instruction pointer (the `rip` register on `x86_64`).

To date, on `x86` *Code Injection Attacks* during which an attacker first introduces payload into the process address space before directly steering control towards this maliciously crafted code are mostly obsoleted by the consistent application of the w⊕x idea.

Consequently, attack methodology has evolved towards so-called *Code Reuse Attacks*. In this type of intrusion, *already existing* code within the program is glued together in order to implement malicious functionality. One concrete shape of a Code Reuse Attack is *Return Oriented Programming* (ROP), a technique where the architectural `x86` stack is set up in a way that chains together so-called *gadgets*. A ROP gadget is an arbitrary sequence of instructions already present in the program that eventually gives control back to an attacker by reading control flow related information from a location controlled by the adversary. An example could be a `ret` instruction reading attacker controlled values from the the stack, but generally ROP can take many different shapes.

Consequently, to protect return addresses saved on the stack, defenders introduced so-called *Stack Canaries*: Contiguous stack-based buffer overflows are detected by checking the validity of magic values (*canaries*) placed at strategic locations on the architectural `x86` stack. Additionally, ASLR (see Section 2.1.2) aims to make the location of data structures in memory unknown to the attacker. Furthermore, in order to further decrease attack surface, defenders introduced Code-Pointer Integrity (CPI) mechanisms such as Pointer Encryption and *relro*.

For the rest of this paper, we will assume all of the aforementioned exploitation mitigation mechanisms to be in place. The authors are aware of the wide corpus of research on this topic, however, this study focuses on the mature mechanisms which were incorporated into nowadays compilers and operating systems.

3 POINTER CLASSIFICATION

In the following, we classify pointers in two stages:

First, we identify all code pointers (pointers pointing into executable memory) that reside in writable memory or in structures referenced by pointers stored in writable memory. We refer to such pointers as *defilable* pointers, because they could be overwritten by an attacker in case a program contains an out-of-bounds write vulnerability. Afterwards, we filter the list of defilable pointers to only include chains ending in code pointers that are eventually used during a control flow transfer (are *live*). For readability reasons, we imply to refer to *live defilable* pointers whenever we use the term *defilable* pointer from here on.

In a second step, we obtain a distance matrix of continuously mapped memory regions within a given process address space containing the relative distances of each region to each other. By performing multiple measurements and determining the entries in the distance matrix we are able to find memory mappings that *even though ASLR is active* are separated by a constant number of bytes across several program invocations. Any pointer residing in a region that has a fixed distance to the region containing the vulnerable array that is accessible out-of-bounds is referred to as *reachable*. Any defilable and reachable pointer can then be used to construct a Wiedergänger-attack.

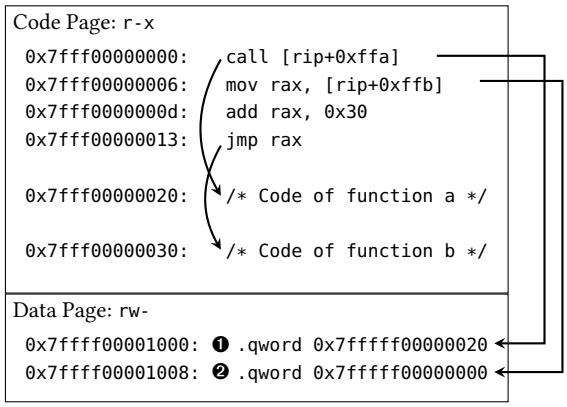

Figure 2: Example of a directly dispatched defilable pointer ❶ and an indirectly dispatched defilable pointer ❷.

In order to keep the attack methodology as independent as possible of the underlying application, we only focus on pointers that are called during *program teardown*. This means that the memory corruption might occur at any point during program execution, but it is only once the program exits that the defiled pointers are dispatched and thus come back to life, exhibiting their malicious behavior[5].

3.1 Identifying defilable pointers

We subclassify defilable pointers into two categories:

First, **directly dispatched defilable pointers** are pointers in writable memory that are read by a control-flow changing instruction. For instance, pointer ❶ in Figure 2 resides at address `0x7ffff00001000` in the data section and is directly referenced as a memory operand by the `call` instruction at address `0x7fff00000000`.

On the other hand, **indirectly dispatched defilable pointers** are pointers in writable memory that are read by a non-control-flow changing instruction but reference data structures which *in turn* contain or reference a pointer that is read by a control-flow changing instruction. In the example shown in Figure 2, pointer ❷ is first read from memory by the `mov` instruction at address `0x7fff00001000` and dispatched later by the `jmp` at address `0x7fff00000013`. Note how the `add` operation modifies the pointer value prior to using it as a jump target. We do not require any such operation when searching for indirectly dispatched defilable pointers, however cases in which an offset is added to a base address come with their own advantages, as discussed later.

[5] i.e. the Wiedergänger returns

3.1.1 Directly Dispatched Defilable Pointers. To allow for a fast systematic search of directly defilable pointers, we build a system that assists us at finding writable pointers in memory, consisting of a tracer and a tracee. In the following we will describe the different steps to perform the systematic search.

(1) At the beginning, the tracer opens a debug handle to the tracee using the `ptrace` debugging API.
(2) Once a special *magic* instruction is executed, the tracee traps into the tracer, and the tracer takes a snapshot of the state of the page tables of the tracee. The magic instruction is used to mark the start of the measurement, and in our scenario typically would be the point where the program starts to terminate. The magic instruction can either be put at instrumentation points of interest, if the source code of the application is available, or be injected using a `LD_PRELOAD` library that wraps library calls of interest.
(3) After obtaining the page table information, the tracer starts injecting `mprotect` calls into the tracee to set all pages with the `write` permission bit active (i.e. `rw-`) to *no access* (i.e. `---`).
(4) Next, the magic instruction is skipped and the tracee is allowed to continue.
(5) Once the tracee tries to access memory that was formerly writable, a segment violation is generated by the operating system kernel, effectively pausing the tracee before control is given to the tracer.
(6) The tracer determines the faulting instruction and checks whether the fault is a read violation caused by an indirect control flow transfer. In this case, the tracer tries to read the memory at the faulting location and checks whether it has the value of a pointer pointing into formerly executable memory. If all conditions are satisfied, the tracer logs the instruction address, the faulting location, as well as the pointer value stored at the faulting location to a file.
(7) The tracer injects another `mprotect` syscall into the tracee to restore the original permissions of the page the tracee is trying to access, and tries to single step over the faulting instruction.
(8) After a successful single step, protection bits are set to *no access* again using a third `mprotect` and execution is allowed to continue.
(9) If any other type of signal is raised by the tracee, the tracer forwards this signal to the tracee in order to establish the original behavior of the debugged application.

Effectively the above procedure sets read watchpoints on all writable locations in the address space. While the `x86` architecture supports watchpoints in hardware by means of special debug registers, they are constrained in number and size [5], resulting in the need of implementing watchpoints simulated in software.

Using this approach, we are able to construct a list of directly dispatched defilable pointers.

3.1.2 Indirectly Dispatched Defilable Pointers. In order to detect indirectly dispatched defilable pointers we use a similar approach to the methodology explained in Section 3.1.1 combined with taint analysis [11]. More precisely, we obtain the desired set of pointers using the following steps:

(1) Step (1) is the same as in Section 3.1.1
(2) Step (2) is the same as in Section 3.1.1
(3) Next, the magic instruction is skipped and the tracee is continued in *single step* mode
(4) After the execution of each instruction, the tracer logs the current state of all registers and the bytes of the current instruction to a file.
(5) Once the tracee exits, a list of *taint sinks* is determined by performing a linear sweep over the traced instruction stream scanning for control flow changing instructions with register or memory operands (such as `call` and `jmp`). Each occurrence of such an instruction type constitutes a taint sink.
(6) Starting from each of the taint sinks, a backwards taint analysis is performed. The goal of this analysis step is to determine the source of the register or memory location read by each sink.
(7) For each sink, additionally the `rdi` register is tainted. This enables us to reason about the source of the first argument of the function targetted by the control flow change and simplifies exploitation later. For instance, an attacker typically wants control flow to call a pointer to the `system` function with the first argument (`rdi`) pointing to the string `"/bin/sh"`.
(8) Taint is propagated following the traced instruction stream backwards using the following rules:
    - Arithmetic operations targetting a tainted register propagate taint to all input registers and keep the target tainted.
    - Any operation belonging to the family of `mov` instructions with register source propagates taint to the input register and removes taint from the destination.
    - Any instruction with tainted destination register using a memory operand as source sanitizes the tainted destination in case of a `mov` instruction. If the source memory operand does not target writable memory, the *base* and the *index* register of the memory operand are tainted, otherwise only the taint on the destination operand is sanitized.
    - If the base register of some source memory operand is the instruction pointer, taint is sanitized in any case, as the instruction pointer is not controllable for a particular instruction located at a particular address.
    - Compare instructions do not taint the flags and are ignored.
    - All control flow changing instructions such as calls, (conditional) jumps, and returns are ignored.
    - All stack-related operations (`push`, `pop`, `leave`) are ignored.
    - Once all taint has been sanitized, or the beginning of the trace is reached, the analysis stops.
(9) Additionally, the instruction pointers of all instructions operating on tainted registers are stored. These sub-traces form the *slices* of the program.

Adhering to this construction, we are able to extract the list of indirectly dispatched defilable pointers as well as all instructions that operate on the pointer value. Due to the rules used in step (8), the whole process yields an over-approximation of the *dynamic backwards slice* of instruction sequences operating on defilable pointers: The fact that compare instructions and conditional

Type	Description
function-local	Local variable on the stack
heap-little	Heap allocation with 128 bytes
heap-big	Heap allocation with 16 MB
thread	Thread Local Storage
global	Global variable
text	Address of executable code
lib-text	Address of `glibc` code
lib-global	`Library global data`

Table 1: List of memory regions that are considered during our measurements.

branches are ignored simplifies away potential range checks that might be performed on pointer values, potentially leading to false-positives that can later be removed using manual analysis. For a similar reason, the taint algorithm also yields pointers that are protected by *glibc*'s Pointer Encryption; Fortunately, due to their unique construction (`ror rX, 0x11; xor rX, fs:0x30`) these are straightforward to recognize and can consequently be filtered out in a following analysis step.

Nevertheless, as discussed in the evaluation, the automated analysis reduces the search space down to a few dozen slices — an amount that can easily be processed manually.

3.2 Identifying reachable pointers

As mentioned before, we now describe how to construct a set of pointers that are located at a fixed offset from user controllable data. Later on, we bootstrap our attack using pointers from the intersection of both sets.

We use a small helper program that allocates different memory types. Table 1 gives a quick overview and a description of the different memory regions that are checked during our test. The following memory types are considered by our program:

Function-Local. Nonstatic function-local memory typically is placed on the architectural `x86` stack, which automatically goes out of scope with the teardown of the respective function. Therefore, pointers to local data structures are assumed to tell us the location of the current stack page.

Thread. When non-local memory that is globally accessible *for one certain thread* but different across all threads is needed, thread local memory is used. This type of memory is incorporated into modern C standards by means of the `__thread` keyword.

Heap-Little / Heap-Big. As some `malloc` implementations allocate memory at different address ranges based on the requested allocation size, we sample pointers returned by malloc for a size of 128 bytes and 16M bytes. For example, `glibc` falls back to using plain `mmap` for requested allocation sizes bigger than `M_MMAP_THRESHOLD` (128K on 64 bit systems) instead of increasing the program break. The obtained pointers are considered to be representative for dynamically allocated memory.

Global. For statically allocated memory, we simply retrieve the address of a global `static` array residing in the `bss` section of the binary with a size of 128 bytes.

Text. Is an address pointing into code that the compiled program consists of. In C this is a function pointer to a function of the program.

Lib-Text. Is an address pointing to code within a shared library. In the concrete test case we use a function pointer referencing `system` in `libc.so`.

Lib-Global. Represents statically allocated memory in a shared library. We use the address of a globally accessible variable within libc to determine the location of this memory type (`stdout`).

In order to determine which pointers have a fixed offset from user controlled data, we execute the helper program multiple times. The helper program outputs addresses of all described memory types, which are then used to calculate a distance matrix of all memory areas to each other. This allows us to determine regions with constant offsets to each other by comparing the distances over multiple executions.

4 EVALUATION

In the following, we collect the results of the tests explained above.

All tests are carried out on binaries compiled with the following protection mechanisms enabled: `-Wl,-z,relro,-z,now -fPIC -pie -fpie -D_FORTIFY_SOURCE=2 -fstack-protector-all` — we think that this configuration reflects best-effort software protections rather well. (For a discussion of *SafeStack* refer to Section 6.1.) The test machine is running the 64 bit version of Arch Linux with kernel version `4.10.6-1`[6] with ASLR in place[7,8]. The `glibc` version in use is 2.25 (February 2017) compiled with *relro* enabled.

4.1 Considered Testcases

As mentioned earlier we only want to focus on defilable pointers that are dispatched during common execution sequences occurring in the C standard library. Therefore we focus on code that is either directly responsible for application teardown (T_0, T_1, T_2, T_3, T_4, T_9, T_{10}) or code that is likely to change the behavior of the code during application teardown (T_5, T_6, T_7, T_8). More specifically, we search for defilable pointers during the following scenarios:

- T_0**: Return from `main`.** This is the most basic way for an application to shutdown. Dynamic and static destructors are dispatched before the application quits.
- T_1**: Call the `exit` function.** Similar to the test above, but also available to functions other than `main` to exit the process.
- T_2**: Call the `_exit` function.** This function is used for immediate shutdown. It simply wraps the respective system call and performs no destructor processing.
- T_3**: Call the `_pthread_exit` function.** Terminates the calling thread and performs destructor handling by calling `exit` if the calling thread is the only thread in the process.
- T_4**: Call the `__stack_chk_fail` function.** This function is usually never called explicitly by any C program. Instead, the

[6] Specifically, we are using Vagrant Box `terrywang/archlinux`, version `3.17.0719`
[7] `/proc/sys/kernel/randomize_va_space` set to 2
[8] `/proc/sys/vm/mmap_rnd_bits` set to 32

compiler inserts code to check the validity of canary values on the stack at the time a protected function returns. If canary validation fails `__stack_chk_fail` is called. The purpose of this test is to simulate a program abort occurring due to a buffer overrun on the stack being detected.

T_5: **Call dynamic memory management functions.** More specifically, we `malloc` (size `0x10`), `realloc` (size `0x20`) and `free` one chunk of memory and then return from main. The rationale behind this is that *glibc* provides hooks[9] that are dispatched on invocation of the dynamic memory allocation related functions (`malloc`, `realloc`, `memalign`, `free`).

T_6: **Register dynamic destructor using `atexit`.** This test registers a destructor at runtime and then returns from `main`. The purpose of this test is to check whether it is possible to defile the newly registered destructor.

T_7: **Register dynamic destructor using `on_exit`.** As the test above but using a different function to register the destructor.

T_8: **Register static destructor.** As the test above but using the `__attribute__((destructor))` function attribute to register a destructor at compile time. This is the classic attack target that `relro` protects against overwriting.

T_9: **Raise a SIGKILL.** This test causes the process to send itself a SIGKILL.

T_{10}: **Violate a heap consistency check.** Similar to test T_4, this is to study what pointers are dispatched during a non-graceful program shutdown that occured due to a violation of a constraint imposed by the heap checker. During the test, we free a `malloc`ed pointer twice to trigger a security abort.

4.2 Directly Dispatched Defilable Pointers

Table 2 shows code pointers in writable memory that are directly dispatched during one or more of our tests T_i, whereas Table 3 gives a more detailed overview of which test case dispatches a particular pointer.

As can be seen from the numbers, pointers D_0 and D_1 are dispatched during 8 of the 11 tests and therefore build the most promising targets to defile. The pairs (D_4, D_0) and (D_5, D_1) dispatch the same location (`dl_rtld_[un]lock_recursive`) but target different functions depending on whether the application depends on `libpthread.so` (the library containing `pthread_exit`). Pointers D_2 and D_3 are called during the creation of a stack trace in case `glibc` detected a security violation. These pointers are special in the sense that they point into code contained in memory that gets allocated *during* the termination process when *glibc* tries to unwind the stack and loads `libgcc_s.so`.

4.3 Indirectly Dispatched Pointers

Table 4 shows the chain of callsites which indirectly dispatch defilable code pointers during one ore more of our tests. The pointer in the middle of the chain can be used as a target during a Wiedergänger attack. The last column indicates whether the pointer is protected using Pointer Encryption (E). Note that even though *relro* places pointers such as the target of I_2 in read-only memory, the taint analysis detects them because the data structures *used by the loader* that eventually reference the location of the destructor are writable.

[9] `man malloc_hook`

#	Callsite (`r-x`) → Pointer Location (`rw-`) → Pointer Target (`r-x`)
D_0	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_lock_recursive` ↪ `ld-2.25.so:rtld_lock_default_lock_recursive`
D_1	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_unlock_recursive` ↪ `ld-2.25.so:rtld_lock_default_unlock_recursive`
D_2	`libc-2.25.so:backtrace_helper` ↪ `libc-2.25.so:unwind_getip` ↪ `libgcc_s.so.1.so:_Unwind_GetIP`
D_3	`libc-2.25.so:backtrace_helper` ↪ `libc-2.25.so:unwind_getcfa` ↪ `libgcc_s.so.1.so:_Unwind_GetCFA`
D_4	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_lock_recursive` ↪ `pthread-2.25.so:pthread_mutex_lock`
D_5	`ld-2.25.so:_dl_fini` ↪ `ld-2.25.so:_rtld_local._dl_rtld_unlock_recursive` ↪ `pthread-2.25.so:pthread_mutex_unlock`

Table 2: Chain of callsites which directly dispatch defilable code pointers for programs using `glibc 2.25` as C standard library. The pointer in the middle of the chain can be used as a target during a Wiedergänger attack. Due to their property of being immediately dispatched from memory, none of the observed pointers D_i is protected by Pointer Encryption or `relro`.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
D_0	2	2			17	2	2	2	2		17
D_1	2	2			17	2	2	2	2		17
D_2					8						8
D_3					8						8
D_4				8							
D_5				8							

Table 3: Directly defilable pointers dispatched during the different test scenarios (numbers are absolute frequencies, no entry means zero)

Table 5 gives a more detailed overview of which test case indirectly dispatches a particular pointer I_i. Pointers I_0, I_1, and I_4 are the direct effects of registering destructors, but are all protected by Pointer Encryption and therefore require the process-specific pointer guard value (`fs:30`) to be known for an attack. As can be seen, seven out of eleven methods to exit a program reach a point in `_dl_fini` where an unprotected defilable pointer gets dispatched indirectly. I_5, I_8 and I_9 are the dynamic memory management related hooks that get called during two tests. The last row in the table indicates a plethora of other potential hooks that we disregarded during manual analysis.

#	Callsite (r-x) → Pointer Location (rw-) → Pointer Target	Encrypted
I_0	libc-2.25.so:__run_exit_handlers ↪ libc-2.25.so:cxafct ↪ ld-2.25.so:_dl_fini	E
I_1	libc-2.25.so:__run_exit_handlers ↪ libc-2.25.so:__libc_atexit ↪ ld-2.25.so:_IO_cleanup	E
I_2	ld-2.25.so:_dl_fini ↪ ld-2.25.so:l->l_info[DT_FINI_ARRAY]->d_un.d_ptr ↪ main_elf:__do_global_dtors_aux_fini_array	—
I_3	ld-2.25.so:_dl_fini ↪ ld-2.25.so:l->l_info[DT_FINI]->d_un.d_ptr ↪ main_elf:_fini	—
I_4	libc-2.25.so:__run_exit_handlers ↪ libc-2.25.so:onfct ↪ main_elf:onexit_dtor	E
I_5	libc-2.25.so:malloc ↪ libc-2.25.so:__malloc_hook_ptr ↪ libc-2.25.so:malloc_hook_ini	—
I_6	libc-2.25.so:_dl_addr ↪ libc-2.25.so:_rtld_global_ptr ↪ ld-2.25.so:__rtld_lock_lock_recursive	—
I_7	libc-2.25.so:_dl_addr ↪ libc-2.25.so:_rtld_global_ptr ↪ ld-2.25.so:__rtld_lock_unlock_recursive	—
I_8	libc-2.25.so:sysmalloc ↪ libc-2.25.so:__morecore_ptr ↪ libc-2.25.so:__morecore	—
I_9	libc-2.25.so:realloc ↪ libc-2.25.so:__realloc_hook_ptr ↪ libc-2.25.so:realloc_hook_ini	—

Table 4: Chain of callsites which indirectly dispatch defilable code pointers for programs using `glibc 2.25` as C standard library.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
I_0	1	1		1		1	2	1	1		
I_1	1	1		1		1	1	1	1		
I_2	1	1		3		1	1	1	2		
I_3	1	1		3		1	1	1	1		
I_4								1			
I_5				1		1					
I_6				2		2					
I_7				2		2					
I_8				2		4					
I_9				1		1					
Other				263	42						42

Table 5: Indirectly dispatched defilable pointers found during the different test scenarios (numbers are absolute frequencies, no entry means zero)

4.4 Reachable Pointers

As we do not assume a primitive to leak memory from the victim program, we need to exploit determinism in the memory allocation strategy used by `mmap` during our attack. This section presents the results obtained when analyzing memory layout of userspace processes. In the following, we restrict our description only to the interesting portion of the results: Memory areas that share a constant offset to each other. Table 6 depicts the results of our measurements on Arch Linux. Every row and column stands for one memory region. Additional labels indicate in what region which memory type is stored. A black field in the table means that the two memory regions have a constant offset to each other. In an optimal ASLR implementation only the diagonal should be visible.

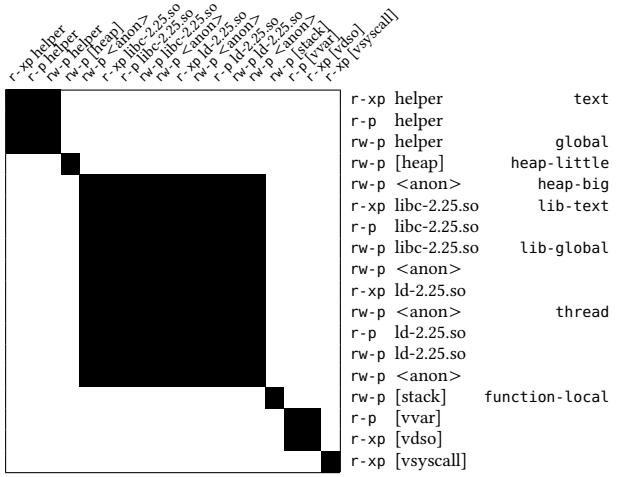

Table 6: Adjacent memory regions on Arch Linux. Black means that two memory regions share a constant offset regardless of the presence of ASLR

The memory layout can be divided into several blocks which are continuously mapped:

program image In Linux the `text`, `data` and `bss` section are always mapped continuously.

heap The heap (program `brk`) is mapped independently.

mmap regions Allocations obtained from `mmap` are also mapped in a continuous way. This means that pointers in any of this regions will give away all other regions in this area.

stack The stack is not in constant distance to any other region.

vvar, vdso These regions are alway next to each other, but independent to the rest of the address space layout.

vsyscall The vsyscall page is always mapped at a constant address.

The most interesting block in the table is the large continuously mapped area caused by Linux' `mmap`. This block contains several potentially user controlled memory types: big heap allocation, all parts of shared library (`text`/`data`/`bss`), and thread local variables. This means that if an attacker finds an unbounded array write they can modify any value in this block by means of a constant offset that only depends on the configuration of the libraries used by the victim. Any defilable pointer in this block potentially enables the attacker to take over the program's control flow.

The reason for this issue is that the userspace loader allocates memory for each new library by making use of the `mmap` system call, which only allocates continuous memory blocks in virtual memory.

Clearly, during our test we find that in current implementations of ASLR on Linux all libraries are loaded in a deterministic manner and thus all relative offsets are constant to each other. For this, once a single address is leaked, the addresses of all functions and

data structures in all libraries are known. Even worse, an attacker who is capable of modifying offsets that are added to defilable pointers prior to dispatching them can attack systems even without a memory leak primitive. We will discuss in Section 5 that this is not a purely theoretical assumption but actually feasible in practice.

5 THE WIEDERGÄNGER-ATTACK

In the following we will give two examples of Wiedergänger-attacks against the dynamic loader and *glibc*. In summary, as mentioned earlier, a Wiedergänger-attack targets global writable function pointers that get dispatched during application teardown and, due to the implementation of ASLR are located at a constant offset to user controlled data. Both examples partially overwrite pointer values to bypass or weaken the effects of ASLR.

5.1 Directly Dispatched Defilable Pointers

Clearly, D_0 and D_1 (D_4, D_5 in multithreaded applications) constitute the most valuable attack targets as they are dispatched in all except two application shutdown scenarios. As shown in the evaluation, these pointers are both defilable, and reachable. Thus, assuming a leak-less exploit, an attacker could launch a Wiedergänger-attack using the corruption technique shown in the code listing in Figure 3. The C listing is the output of a script that automatically constructs a basic attack for demonstration purposes, which can be reproduced using the package given in Section 9.

```
// [+] argument for wiedergaenger attack is at ld+0x224948
// [+] target for wiedergaenger attack is at ld+0x224f48
// [+] system is at 0x3f450 in libc
// [+] constant offset between mmaped chunk and ldbase is 0x59fff0

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    unsigned char *ptr;
    ptr = malloc(0x200000);
    printf("%p\n", ptr);

    /* Defiling pointers */
    ptr[0x7c4938] = '/'; ptr[0x7c4939] = 'b'; ptr[0x7c493a] = 'i';
    ptr[0x7c493b] = 'n'; ptr[0x7c493c] = '/'; ptr[0x7c493d] = 's';
    ptr[0x7c493e] = 'h';

    ptr[0x7c4f38] = 0x50; ptr[0x7c4f39] = 0x94; ptr[0x7c4f3a] = 0xa7;

    /* The program continues until it eventually exits */
    return 0;
}
```

Figure 3: Minimal example of a Wiedergänger-attack spawning a shell on Debian Buster with a probability of 1:4096 by using the directly dispatched pointer D_0.

To understand the code shown in Figure 3, consider the program slice belonging to D_0 found in `_dl_fini` in `ld.so` during our evaluation (all pointer values are examples for one particular run and affected by ASLR):

```
/* Points to 0x7ffff7ffd948 (writable) */
0x7ffff7de8d1f:    lea   rdi, qword ptr [rip + 0x214c22]
/* Points to 0x7ffff7ffdf48 (writable) */
0x7ffff7de8d26:    call  qword ptr [rip + 0x21521c]
```

Additionally, assume the pointer value returned by `malloc` in Figure 3 is `0x7ffff7839010`, and the `system` function is located at `0x7ffff7a79450`.

Then the (constant) distances of the pointer returned by `malloc` to the two addresses used as targets for the `lea` and `call` instructions in D_0 are `0x7ffff7ffd948 - 0x7ffff7839010 = 0x7c4938` and `0x7ffff7ffdf48 - 0x7ffff7839010 = 0x7c4f38`. Note how these distances can be (independently from ASLR) calculated a priori and are used as out-of-bounds array indices in Figure 3. The first write sequence sets up the string `/bin/sh` whereas the second write sequence performs a three byte partial override of the pointer stored at `0x7ffff7ffdf48`. The byte sequence `50 94 a7` corresponds to the three least significant bytes of the `system` function. Thus, the code above will execute `system("/bin/sh")` resulting in arbitrary code execution in context of the attacked process.

Discussing the same example with ASLR taken into account, we directly see that even though all absolute pointer values change, the values used as indices for the array remain the same (as they were found during the reachable analysis). The only point where the attack uses an absolute address is the three-byte-override. As discussed earlier, ASLR is performed at page granularity. This means that out of 24 overwritten bits (three bytes), twelve bits remain constant, leaving an attacker with twelve unknown bits. This results in an attack probability of $1 : 2^{12} = 1 : 4096$ in the worst case.

In the next section, we will remedy the $1 : 2^{12}$ attack probability using indirectly dispatched pointers to achieve reliable exploitation.

5.2 Indirectly Dispatched Pointers

To achieve reliable code execution with a Wiedergänger-attack, we make use of the instruction sequence of I_3 with `rbx` pointing to writable memory. The Assembler instructions are shown in Figure 4a. The C source code equivalent (Figure 4b) can be found in the *glibc* source in `dl-fini.c` in function `_dl_fini`.

When entering the Wiedergänger-gadget I_3 from above, `rbx` holds the address of a `struct link_map` referencing control data used by the dynamic loader. This `struct` in turn contains three relevant elements: (1) the base address `l_addr` of the main executable ELF file at offset `0x0` (corresponding to `[rbx + 0x0]` in the ASM listing), (2) the pointer `l_info[DT_FINI_ARRAYSZ]` to the size of the `FINI_ARRAY` of the main executalbe ELF file at offset `0x120` (`[rbx + 0x120]`), and (3) the pointer `l_info[DT_FINI]` holding a pointer to the offset of the `.fini` destructor to the base address of the main ELF executable at offset `0xa8` (`[rbx + 0xa8]`).

To achieve reliable exploitation we abuse the fact that the code performs an *addition* to calculate the absolute address of the `.fini` function in the last line of the C listing. As explained, the pointer `l_info[DT_FINI]` usually points to the *offset* of the `.fini` function

```
mov    r12, qword ptr [rax + 8]
mov    rax, qword ptr [rbx + 0x120]
add    r12, qword ptr [rbx]
mov    rdx, qword ptr [rax + 8]
shr    rdx, 3
test   edx, edx
lea    r15d, dword ptr [rdx - 1]
jne    loc_a
jmp    loc_b
loc_a:
mov    edx,r15d
call   qword ptr [r12 + rdx * 8]
/* ^ This call instruction is never reached during exploitation. */
/* As r12 contains a negative number wrongly interpreted by the  */
/* call as a pointer to kernel memory it would result in a crash */
loc_b:
mov     rax, qword ptr [rbx + 0xa8]
mov     rax, qword ptr [rax + 8]
add     rax, qword ptr [rbx]
call    rax
```

(a) Assembly Listing of the Program Slice belonging to I_3

```
struct link_map *l = maps[i];
/* ... */

/* First see whether an array is given.  */
if (l->l_info[DT_FINI_ARRAY] != NULL)
  {
    ElfW(Addr) *array =
      (ElfW(Addr) *) (l->l_addr
              + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
    unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
              / sizeof (ElfW(Addr)));
    while (i-- > 0)
      ((fini_t) array[i]) ();
  }

/* Next try the old-style destructor.  */
if (l->l_info[DT_FINI] != NULL)
  DL_CALL_DT_FINI
    (l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
```

(b) C Listing of the Source Code Corresponding to I_3

Figure 4: Indirectly dispatched defilable pointer I_3 that allows to spawn a shell by (partially) overwriting `l->l_addr` and `l->l_info[DT_FINI]` to form the address of a *win-gadget* dispatched in the last line of the C listing.

within the main ELF executable. However, *close* to this information, the loader places an absolute pointer to the variable `_r_debug` in `ld.so`. Consequently it becomes possible to overwrite the least significant byte of `l_info[DT_FINI]` and *let it point to an absolute adress* (`_r_debug`, randomized by ASLR). Then, `l->l_addr` can be overwritten with the constant distance of `_r_debug` to a so-called *win*-gadget[10] in *glibc* that executes `execve("/bin/bash")`. The central idea that lets the attack succeed is to *exchange base address and offset* during calculation of the destructor's location, with `l->l_info[DT_FINI]` becoming a pointer to a pointer, and `l->l_addr` being a constant offset.

The technique outlined above, however, needs to overcome one more problem: If `l->l_addr` does not hold a valid base address anymore the `array` variable in the C source code listing will be assigned an invalid pointer that will result in a crash when being dispatched in the last line of the first if block. To remedy this, we use another one-byte override to corrupt the pointer `l_info[DT_FINI_ARRAYSZ]` and let it point to any value that is smaller than `sizeof(ElfW(Addr)) = 8` such that the integer division used to compute the variable `i` becomes zero. Fortunately, there are several such values close to the original pointer value of `l_info[DT_FINI_ARRAYSZ]`.

Combining all of this, Figure 5 in the Appendix shows the example of a C program that corrupts the loader's internal data structures in the way outlined above to spawn a shell. As this Wiedergänger-attack only uses constant offsets and one-byte overrides, exploitation succeeds reliably for a known combination of main executable, dynamic loader and all shared library dependencies. A visualization of the attack carried out by the code depicted in Figure 5 can be found in the Appendix in Figure 6.

We would like to point out that both pointers targeted by the partial overrides usually point into the `.dynamic` section of the main ELF executable and therefore are (a) known to an attacker who is in possesion of the binary and (b) unlikely to change as they are part of the ELF specification.

[10] https://github.com/david942j/one_gadget

6 DISCUSSION

In this section we discuss why the Wiedergänger-attack is possible on current Linux systems, and how possible mitigation strategies might look like.

6.1 On the Effectiveness of Current Security Mechanisms

In this section, we discuss how Wiedergänger relates to currently employed security mechanisms.

ASLR. The basic idea of ASLR is to hide the addresses of library code within the program from an attacker in order to hinder the efficient detection of ROP gadgets. However, as the heap region containing large allocations consisting of user controlled data has a constant distance from library code, such as the `libc`, an attacker is again able to calculate the memory addresses of the required Wiedergänger-gadgets. This issue effectively defeats ASLR in practice.

SafeStack. SafeStack [8, 12] is a newer protection mechanism to mitigate stack-based buffer overflows. SafeStack splits the program's stack into the safe and unsafe stack. The safe stack is used like the normal stack for storing control relevant data like return address and local variables that are considered safe. The unsafe stack is used as a storage place for anything that could overflow and corrupt the stack in normal unprotected programs.

In context of our work, SafeStack is an interesting mechanism since it uses `mmap` to allocate the second (unsafe) stack. So far we concluded that user controlled data on the stack (function-local) does not allow an attacker to reach defilable pointers. Under Linux

the stack is located at an random address and has no constant offset to any other section in the memory layout.

In a program protected by the SafeStack mechanism this property only holds for the safe stack. The unsafe stack, which contains only potentially unsafely accessed data, is allocated like any other `mmap` region and hence could be used by an attacker to reach defilable pointers.

This property means that a program with an unbounded array access vulnerability with the array on the stack that would not be exploitable with Wiedergänger becomes exploitable by enabling an additional security mechanism. This has the ironic consequence that *SafeStack potentially makes programs less secure.*

6.2 Attack Mitigation Strategies

Interestingly, there already exists a patch that allows to allocate memory at properly randomized virtual addresses [10]. This patch, while being discussed, never made it into the upstream sources for unknown reasons. With the introduction of Wiedergänger-attacks, the need for ASLR producing memory ranges with non-constant offsets becomes even more evident.

Protecting writable pointers is not so straightforward, unfortunately. As a first step, we propose to build pointer mangling into the compiler toolchain, to take away the burden of enciphering and deciphering pointers from the programmer. This however has the unfortunate consequence of making externally visible writable pointers (such as for example the `free_hook`) part of the application binary interface. This is for the simple reason that any application potentially can overwrite the pointer for legitimate purposes, and only if both, the library providing the hook, as well as the overwriting application are compiled with the same pointer mangling settings, functionality can be guaranteed.

6.3 Future Work

In our investigation we have only looked at small, single threaded applications. There are two different aspects that we also want to investigate in future work. First, we want to combine our approach with a tool that increases the code coverage, like for example the fuzzing tool *AFL* [15]. With this, we intend to not only detect general hooks provided by the standard library or the application loader, but also application specific hooks that may be exploited by an attacker.

Second, our proof-of-concept implementation currently is only capable to follow one thread of execution. In order to extract a more holistic view, we intend to extend our framework to follow multiple threads.

Third, we would like to repeat our brief case study on a wider set of C standard libraries running on a wider range of operating systems.

Last, while we have shown that some types of memory get allocated at predictable locations, is is unclear how predictability of the sequence of allocations affects the applicatbility of Wiedergänger. For example, a web browser allocates different objects at run-time depending on user interaction. A more thorough study analyzing allocation patterns in large compiled object oriented programs is therefore left as future work.

7 RELATED WORK

Dynamic Hooks. Dynamic hooks, a concept similar to the hooks leveraged by Wiedergänger, are presented by Vogl et al. [13]. In their work, the authors modify transient control data in the Linux kernel and modify non-control flow related data structures while exploiting kernel vulnerabilities during runtime (i.e. when the data is used by the kernel). To realize their idea their prototype also *makes use of static program slicing and symbolic execution to automatically extract paths for dynamic hooks that can then be used by a human expert for their realization* [13].

Similar to their approach, we apply dynamic taint analysis and backward slicing to detect hooks in userspace program code. In contrast to Vogl's work, we modify control flow relevant data, such as pointers or offsets that are later added to a given base pointer by the application. With this, our approach is able to defeat ASLR, as our approach leverage pointers that are used with an (writable) offset.

Bypassing ASLR. In our work we use the layout and architecture of the Linux dynamic loader and the standard library for offensive purposes. Other work is also concerned with leveraging the layout of the binary format (ELF for Linux). Leakless, for example, uses the dynamic loader's functionality to resolve library functions during runtime in order to break ASLR without the need of an address leak. With this they effectively eliminate the information leak step that is typically required during exploitation [7]. Similarly, we leverage hooks within the program loader and inside *glibc* that are present in the memory of every program executed. With this, our approach also gains generality, as all applications that use the dynamic loader and *glibc* as standard library are attackable.

Marco-Gisbert and Ripoll [9] have found a related problem in Linux' memory management. In their work, they point out that the application code used to be placed at a static offset to library code (*offset2lib* vulnerability). For this, when leaking the address of the executable code, an attacker is able to calculate the addresses of library code and vice versa. As a result `mmap` was modified to randomize the code section of the executable binary independently of the rest of the virtual memory. However, in this work, we show that the original problem still persists: Due to the allocation strategy for both library code and data as well as the heap data structures for big objects are still allocated in an adjacent block. Thus, when leaking the address of the code or data of one library or of an object on the heap, an attacker is still able to calculate the addresses of the code and data of other libraries.

Taint Analysis. Finally, taint analysis on its own is an entire field of research. To extract existing hooks during program teardown, we make use of taint analysis. As already previously noted, our approach over-approximates the set of indirectly defilable pointers and thus requires a human analyst to further filter out false positives. This limitation could however be avoided by extending our implementation to use enhanced *Bit-level Taint Analysis* [14]. In this work, we however refrained from this technique due to the higher implementation effort.

Control Flow Integrity. In its current form, the Wiedergänger-attack modifies code pointers or offsets that are added to an existing base address during runtime. Admittedly this type of attack is easily

detectable when fine-grained forward edge Control Flow Integrity (CFI) mechanisms (like proposed by Abadi [1]) are employed. The problem with CFI however is that no such implementation currently exists for off-the-shelf usage. Existing mechanisms (or mechanisms that will most likely find large adoption, like Intel CET [4]) do not restrict the forward edges enough and are thus still unable to detect Wiedergänger-attacks that aim to call original function entry points. This is due to the fact that these mechanisms only maintain a small number of sets of different allowed call targets. Intel CET even only maintains one global set such that every possible function is still callable.

8 CONCLUSION

We have introduced the Wiedergänger-attack, a new attack vector targeting C programs running on Linux with `glibc`. To separate corruption and exploitation time, we introduce the notion of a defiled pointer, which is a code pointer located in writable memory. Defiled pointers can reside within the program without affecting their behaviour during normal operation; instead they are dispatched by the C runtime environment during program shutdown, bringing the malicious payload to live only instructions before the regularly scheduled program's death. The main reason defiling becomes possible because `mmap` does not provide proper randomization strategies.

We think that Wiedergänger-attacks become the most powerful when combined together with other bugs, such as information disclosure bugs, but they also put in question the currently used randomization strategy employed in current Linux systems. After all, with the possibility of gaining arbitrary code execution by only using constant information that can be obtained by an attacker regardless of the state of ASLR, we would like to encourage future research on attacks against the dynamic loader (Loader Oriented Programming).

9 AVAILABILITY

To encourage open research, we distribute all tools and measurements as well as a sample script that bootstraps a simple Wiedergänger-attack against any Linux system running on `x86-64` under an open source license. The package can be found on the project's website

https://kirschju.re/projects/wiedergaenger .

ACKNOWLEDGEMENTS

The research was supported by the German Federal Ministry of Education and Research under grant 16KIS0327 (IUNO).

REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-Flow Integrity. In *Conference on Computer and Communications Security (CCS)* (2005).
[2] Aleph One. Smashing the Stack for Fun and Profit. *Phrack 7*, 49 (1996).
[3] Corporation, I. 5-Level Paging and 5-Level EPT. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, May 2017.
[4] Corporation, I. Control-flow enforcement technology preview. Tech. rep., Intel Corporation, 2017.
[5] Corporation, I. Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1, 2017.
[6] Drepper, U. Pointer Encryption. http://udrepper.livejournal.com/13393.html, Jan. 2007. Accessed 2017-07-24 18:09.
[7] Federico, A. D., Cama, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. How the ELF ruined christmas. In *USENIX Security Symposium* (Washington, D.C., 2015), USENIX Association, pp. 643–658.
[8] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 147–163.
[9] Marco-Gisbert, H., and Ripoll, I. On the effectiveness of full-aslr on 64-bit linux. In *INDePth Security Conference, DeepSec* (11 2014).
[10] Roberts, W. [RFC] Introduce mmap randomization. https://patchwork.kernel.org/patch/9248669/, 2016.
[11] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy* (2010).
[12] The Clang Team. Clang 5 Documentation. https://clang.llvm.org/docs/SafeStack.html, Feb. 2017.
[13] Vogl, S., Gawlik, R., Garmany, B., Kittel, T., Pfoh, J., Eckert, C., and Holz, T. Dynamic hooks: Hiding control flow changes within non-control data. In *USENIX Security Symposium* (Aug. 2014), USENIX.
[14] Yadegari, B., and Debray, S. *Bit-level taint analysis.* Institute of Electrical and Electronics Engineers Inc., 12 2014, pp. 255–264.
[15] Zalewski, M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2015.

APPENDIX

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    unsigned char *ptr;
    ptr = malloc(0x200000);

#if __GLIBC__ == 2 && __GLIBC_MINOR__ == 24

    /*********************************************/
    /* Debian Buster kernel 4.12.6-1 (glibc 2.24-17) */
    /*********************************************/

    /* Distance of the malloced pointer and the struct link_map */
    /* used by ld */
    unsigned long base = 0x7c3160;

    /* Set l->l_addr to fixed offset of _r_debug in ld.so and */
    /* a win-gadget in libc.so */
    *(unsigned long long *)&ptr[base] = 0xfffffffffb1480f;

    /* Set l->l_info[DT_FINI] pointer to a pointer to _r_debug */
    ptr[base +  0xa8] = 0xb8;

    /* Set l->l_info[DT_FINI_ARRAYSZ] pointer to a value < 8 */
    ptr[base + 0x120] = 0xc0;

#endif
#if __GLIBC__ == 2 && __GLIBC_MINOR__ == 25
    /*********************************************/
    /* Arch Linux kernel 4.10.6-1 (glibc 2.25) */
    /*********************************************/

    /* Distance of the malloced pointer and the struct link_map */
    /* used by ld */
    unsigned long base = 0x7c90e0;

    /* Set l->l_addr to fixed offset of _r_debug in ld.so and */
    /* a win-gadget in libc.so */
    *(unsigned long long *)&ptr[base] = 0xfffffffffb11453;

    /* Set l->l_info[DT_FINI] pointer to a pointer to _r_debug */
    ptr[base +  0xa8] = 0xa8;

    /* Set l->l_info[DT_FINI_ARRAYSZ] pointer to a value < 8 */
    ptr[base + 0x120] = 0x60;
#endif

    return 0;
}
```

Figure 5: Minimal example of a reliable Wiedergänger-attack spawning a shell on Debian 10 (*glibc 2.24*) and Arch Linux (*glibc 2.25*) using indirectly dispatched pointer I_3 and 1-byte partial pointer overwrites to bypass ASLR. Note that all numbers are constant, even in presence of ASLR.

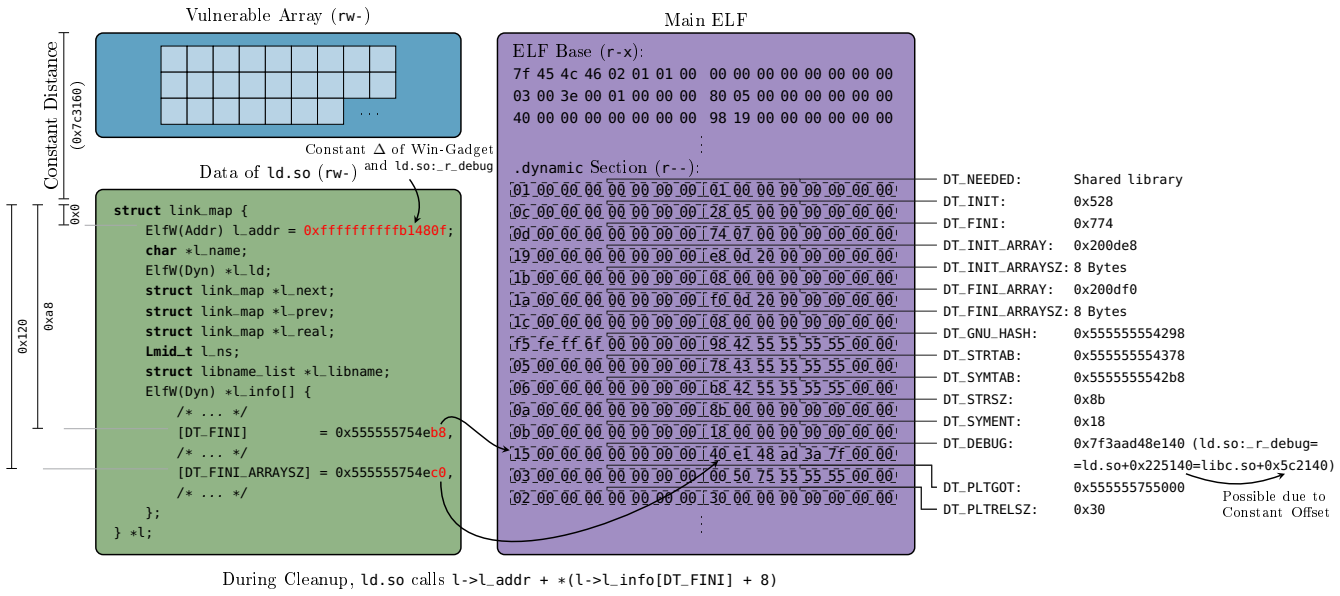

Figure 6: Visualization of the reliable Wiedergänger-attack spawning a shell on Debian 10 (*glibc 2.24*). The graphic depicts the changed members of `struct link_map` in red.

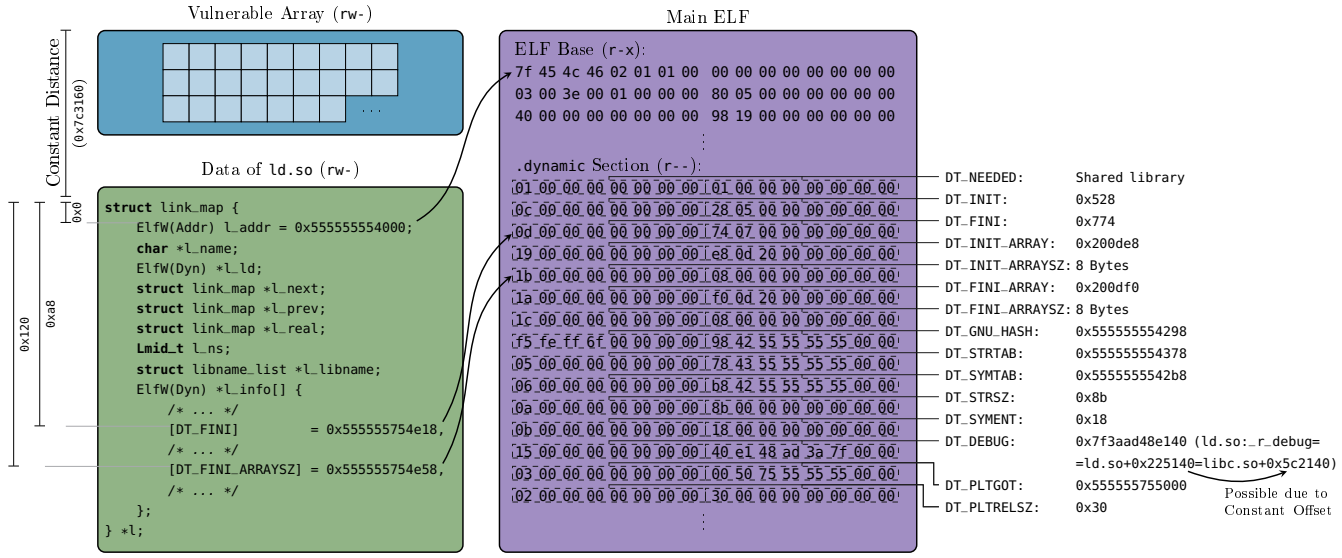

Figure 7: Visualization of the pointer values cointained in `struct link_map` during normal program execution