usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

How the ELF Ruined Christmas

Alessandro Di Federico, *University of California, Santa Barbara and Politecnico di Milano;*
Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna,
University of California, Santa Barbara

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/di-frederico

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of
the 24th USENIX Security Symposium
is sponsored by USENIX

How the ELF Ruined Christmas

Alessandro Di Federico[1,2], Amat Cama[1], Yan Shoshitaishvili[1], Christopher Kruegel[1], and Giovanni Vigna[1]

[1]University of California, Santa Barbara, CA, USA
{amat,yans,chris,vigna}@cs.ucsb.edu
[2]Politecnico di Milano, Milan, Italy
alessandro.difederico@mail.polimi.it

Abstract

Throughout the last few decades, computer software has experienced an arms race between exploitation techniques leveraging memory corruption and detection/protection mechanisms. Effective mitigation techniques, such as Address Space Layout Randomization, have significantly increased the difficulty of successfully exploiting a vulnerability. A modern exploit is often two-stage: a first information disclosure step to identify the memory layout, and a second step with the actual exploit. However, because of the wide range of conditions under which memory corruption occurs, retrieving memory layout information from the program is not always possible.

In this paper, we present a technique that uses the dynamic loader's ability to *identify* the locations of critical functions directly and call them, without requiring an information leak. We identified several fundamental weak points in the design of ELF standard and dynamic loader implementations that can be exploited to resolve and execute arbitrary library functions. Through these, we are able to bypass specific security mitigation techniques, including partial and full RELRO, which are specifically designed to protect ELF data-structures from being coopted by attackers. We implemented a prototype tool, Leakless, and evaluated it against different dynamic loader implementations, previous attack techniques, and real-life case studies to determine the impact of our findings. Among other implications, Leakless provides attackers with reliable and non-invasive attacks, less likely to trigger intrusion detection systems.

1 Introduction

Since the first widely-exploited buffer overflow used by the 1998 Morris worm [27], the prevention, exploitation, and mitigation of memory corruption vulnerabilities have occupied the time of security researchers and cybercriminals alike. Even though the prevalence of memory corruption vulnerabilities has finally begun to decrease in recent years, classic buffer overflows remain the third most common form of software vulnerability, and four other memory corruption vulnerabilities pad out the top 25 [13].

One reason behind the decreased prevalence of memory corruption vulnerabilities is the heavy investment in research on their prevention and mitigation. Specifically, many mitigation techniques have been adopted in two main areas: system-level hardening (such as CGroups [23], AppArmor [4], Capsicum [41], and GRSecurity [18]) and application-level hardening (such as stack canaries [3], Address Space Layout Randomization (ASLR), and the *No-eXecute* (NX) bit [8]).

In particular, *Address Space Layout Randomization* (ASLR), by placing the dynamic libraries in a random location in memory (unknown to the attacker), lead attackers to perform exploits in two stages. In the first stage, the attacker must use an *information disclosure* vulnerability, in which information about the memory layout of the application (and its libraries) is revealed, to identify the address of code that represents security-critical functionality (such as the `system()` library function). In the second stage, the attacker uses a *control flow redirection* vulnerability to redirect the program's control flow to this functionality.

However, because of the wide range of conditions under which memory corruptions occur, retrieving this information from the program is not always possible. For example, memory corruption vulnerabilities in parsing code (e.g., decoding images and video) often take place without a direct line of communication to an attacker, precluding the possibility of an information disclosure. Without this information, performing an exploit against ASLR-protected binaries using current techniques is often infeasible or unreliable.

As noted in [36], despite the race to harden applications and systems, the security of some little-known aspects of application binary formats and the system components using them, have not received much scrutiny. In particular we focus on the *dynamic loader*, a userspace component of the operating system, responsible for loading binaries, and the libraries they depend upon, into memory. Binaries use the dynamic loader to support the *resolution* of imported symbols. Interestingly, this is the exact behavior that an attacker of a hardened application attempts to reinvent by leaking a library's address and contents.

Our insight is that a technique to eliminate the need for an information disclosure vulnerability could be developed by abusing the functionality of the dynamic loader. Our technique leverages weaknesses in the dynamic loader and in the general design of the ELF format to resolve and execute arbitrary library functions, allowing us to successfully exploit hardened applications without the need for an information disclosure vulnerability. Any library function can be executed with this technique, even if it is not otherwise used by the exploited binary, as long as the library that it resides in is loaded. Since almost every binary depends on the C Library, this means our technique allows us to execute security-critical functions such as `system()` and `execve()`, allowing arbitrary command execution. We will also show application-specific library functions can be re-used to perform sophisticated and stealthy attacks. The presented technique is reliable, architecture-agnostic, and does not require the attacker to know the version, layout, content, or any other unavailable information about the library and library function in question.

We implemented our ideas in a prototype tool, called Leakless[1]. To use Leakless, the attacker must possess the target application, and have the ability to exploit the vulnerability (i.e., hijack control flow). Given this information, Leakless can automatically construct an exploit that, without the requirement of an information disclosure, invokes one or more critical library functions of interest.

To evaluate our technique's impact, we performed a survey of several different distributions of Linux (and FreeBSD) and identified that the vast majority of binaries in the default installation of these distributions are susceptible to the attack carried out by Leakless, if a memory corruption vulnerability is present in the target binary. We also investigated the dynamic loader implementations of various C Libraries, and found that most of them are susceptible to Leakless' techniques. Additionally, we showed that a popular mitigation technique, RELocation Read-Only (RELRO), which protects library function calls from being redirected by an attacker, is completely bypassable by Leakless. Finally, we compared the length of Leakless' ROP chains against ROP compilers implementing similar functionality. Leakless produces significantly shorter ROP chains than existing techniques, which, as we show, allows it to be used along with a wider variety of exploits than similar attacks created by traditional ROP compilers.

In summary, we make the following contributions:

- We develop a new, architecture- and platform-agnostic attack, using functionality inherent in ELF-based system that supports dynamic loading, to enable an attacker to execute arbitrary library functions without an information disclosure vulnerability.
- We detail, and overcome, the challenges of implementing our system for different dynamic loader implementations and in the presence of multiple mitigation techniques (including RELRO).
- Finally, we perform an in-depth evaluation, including a case study of previously complicated exploits that are made more manageable with our technique, an assessment of the security of several different dynamic loader implementations, a survey of the applicability of our technique to different operating system configurations, and a measurement of the improvement in the length of ROP chains produced by Leakless.

2 Related Work: The Memory Corruption Arms Race

The memory corruption arms race (i.e., the process of defenders developing countermeasures against known exploit techniques, and attackers coming up with new exploitation techniques to bypass these countermeasures) has been ongoing for several decades. While the history of this race has been documented elsewhere [37], this section focuses on the sequence of events that has required many modern exploits to be *two-stage*, that is, needing an *information disclosure* step before an attacker can achieve arbitrary code execution.

Early buffer overflow exploits relied on the ability to inject binary code (termed *shellcode*) into a buffer, and overwrite a return address on the stack to point into this buffer. Subsequently, when the program would return from its current function, execution would be redirected to the attacker's shellcode, and the attacker would gain control of the program.

As a result, security researchers introduced another mitigation technique: the *NX* bit. The NX bit has the effect of preventing memory areas not supposed to contain code (typically, the stack) from being executed.

The *NX bit* has pushed attackers to adapt the concept of *code reuse*: using functionality already in the program (such as system calls and security-critical library functions) to accomplish their goals. In return-into-libc exploits [30, 39], an attacker redirects the control flow directly to a sensitive libc function (such as `system()`) with the proper arguments to perform malicious behavior, instead of using injected shellcode.

To combat this technique, a system-level hardening technique named *Address Space Layout Randomization*

[1] The source code is available at: `https://github.com/ucsb-seclab/leakless`

(ASLR) was developed. When ASLR is in place, the attacker does not know the location of libraries, in fact, the program's memory layout (the locations of libraries, the stack, and the heap) is randomized at each execution. Because of this, the attacker does not know *where* in the library to redirect the control flow in order to execute specific functions. Worse, even if the attacker is able to determine this information, he is still unable to identify the location of specific functions inside the library unless he is in possession of a copy of the library. As a result, an attacker usually has to leak the contents of the library itself and parse the code to identify the location of critical functions. To leak these libraries, attackers often reuse small chunks of code (called *gadgets*) in the program's code segment to disclose memory locations. These gadgets are usually combined by writing their addresses onto the stack and consecutively returning to them. Thus, this technique is named *Return Oriented Programming* (ROP) [35].

ROP is a powerful tool for attackers. In fact, it has been shown that a "Turing-complete" set of ROP gadgets can be found in many binaries and can be employed, with the help of a *ROP compiler*, to carry out exploitation tasks [34]. However, because of their generality, ROP compilers tend to produce long ROP chains that, depending on the specific details of a vulnerability, are "too big to be useful" [22]. Later, we will show that Leakless produces relatively short ROP chains, and, depending on present mitigations, requires very few gadgets. Additionally, Leakless is able to function without a Turing-complete gadget set.

In real-world exploits, an attacker usually uses an *information disclosure* attack to leak the address or contents of a library, then uses this information to calculate the correct address of a security-critical library function (such as `system()`), and finally sends a second payload to the vulnerable application that redirects the control flow to call the desired function.

In fact, we observed that that the goal of finding the address of a specific library function is actually already implemented by the *dynamic loader*, an OS component that facilitates the resolution of dynamic symbols (i.e., determining the addresses of library functions). Thus, we realized that we could leverage the dynamic loader to remove the information disclosure step, and craft exploits, which would work without the need of an information disclosure attack. Since our attack does not require an information leak step, we call it *Leakless*.

The concept of using the dynamic loader as part of the exploitation process was briefly explored in the context of return-into-libc attacks [15,21,30]. However, existing techniques are extremely situational [30], platform-dependent, require two stages [21], or are susceptible to current mitigation techniques such as RELRO [30], which we will discuss in future sections. Leakless, on the other hand, is a single-stage, platform-independent, general technique, and is able to function in the presence of such mitigations.

In the next section, we will describe how the dynamic loader works, and afterwards will show how we abuse this functionality to perform our attack.

3 The Dynamic Loader

The dynamic loader is a component of the userspace execution environment that facilitates loading the libraries required by an application at start time and resolving the dynamic symbols (functions or global variables) that are exported by libraries and used by the application. In this section, we will describe how dynamic symbol resolution works on systems based on the ELF binary object specification [33].

ELF is a standard format common to several Unix-like platforms, including GNU/Linux and FreeBSD, and is defined independently from any particular dynamic loader implementation. Since Leakless mostly relies on standard ELF features, it is easily applicable to a wide range of systems.

3.1 The ELF Object

An application comprises a main binary ELF file (the executable) and several dynamic libraries, also in ELF format. Each ELF object is composed of *segments*, and each segment holds one or more *sections*.

Each section has a conventional meaning. For instance, the `.text` section contains the code of the program, the `.data` section contains its writeable data (such as global variables), and the `.rodata` section contains the read-only data (such as constants and strings). The list of sections is stored in the ELF file as an array of `Elf_Shdr` structures.

Note that there are two versions of each ELF structure: one version for 32-bit ELF binaries (e.g., `Elf32_Rel`) and one for 64-bit (e.g., `Elf64_Rel`). We ignore this detail for the sake of simplicity, except in specific cases where it is relevant to our discussion.

3.2 Dynamic Symbols and Relocations

In this section, we will give a summary of the data structures involved in ELF symbol resolution. Figure 1 gives an overview of these data structures and their mutual relationships.

An ELF object can export symbols to and import symbols from other ELF objects. A symbol represents a function or a global variable and is identified by a name.

Each symbol is described by a corresponding `Elf_Sym` structure. This structure, instances of which comprise the `.dynsym` ELF section, contains the following fields relevant to our work:

`st_name`. An offset, relative to the start of the `.dynstr` section, where the string containing the name of the symbol is located.

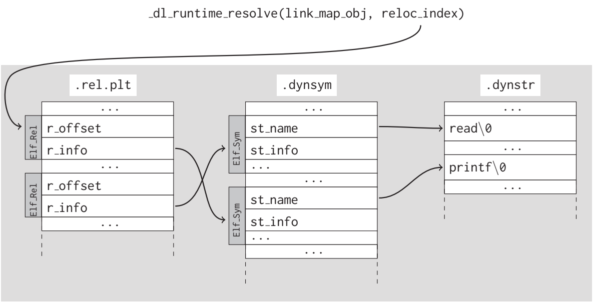


Figure 1: The relationship between data structures involved in symbol resolution (without symbol versioning). Shaded background means read only memory.

st_value. If the symbol is exported, the virtual address of the exported function, NULL otherwise.

These structures are referenced to resolve imported symbols. The resolution of imported symbols is supported by relocations, described by the Elf_Rel structure. Instances of this structure populate the .rel.plt section (for imported functions) and the .rel.dyn section (for imported global variables). In our discussion we are only interested to the former section. The Elf_Rel structure has the following fields:

r_info. The three least significant bytes of this field are used as an unsigned index into the .dynsym section to reference a symbol.

r_offset. The location (as an absolute address) in memory where the address of the resolved symbol should be written to.

When a program imports a certain function, the linker will include a string with the function's name in the .dynstr section, a symbol (Elf_Sym) that refers to it in the .dynsym section, and a relocation (Elf_Rel) pointing to that symbol in the .rel.plt section.

The target of the relocation (the r_offset field of the Elf_Rel struct) will be the address of an entry in a dedicated table: the Global Offset Table (GOT). This table, which is stored in the .got.plt section, is populated by the dynamic loader as it resolves the relocations in the .rel.plt section.

3.3 Lazy Symbol Resolution

Since resolving every imported symbol and applying all relocations at application startup can be a costly operation, symbols are resolved *lazily*. In lazy symbol resolution, the address of a function (which corresponds to an entry in the GOT) is only resolved when necessary (i.e., the first time the imported function is called).

When a program wants to calls an imported function, it instead calls a dedicated stub of code, located in the Procedure Linkage Table (the .plt section). As shown in Listing 1, each imported function has a stub in the PLT that performs an unconditional indirect jump to the associated entry in the GOT.

After symbol resolution, this GOT entry contains the address of the actual function, in the imported library, and execution continues seamlessly into this function. When the function returns, control flow returns to the *caller* of the PLT stub, and the rest of the PLT stub is not executed. However, at program startup, GOT entries are initialized with an address pointing to the *second* instruction of the associated PLT stub. This part of the stub will push onto the stack an identifier of the imported function (in the form of an offset to an Elf_Rel instance in the .rel.plt section) and jump to the PLT0 stub, a piece of code at the beginning of the .plt section. In turn, the PLT0 stub, pushes the value of GOT[1] onto the stack and performs an indirect jump to the address of GOT[2]. These two entries in the GOT have a special meaning and the dynamic loader populates them at application startup:

GOT[1]. A pointer to an internal data structure, of type link_map, which is used internally by the dynamic loader and contains information about the current ELF object needed to carry out symbol resolution.

GOT[2]. A pointer to a function of the dynamic loader, called _dl_runtime_resolve.

In summary, PLT entries basically perform the following function call:

```
_dl_runtime_resolve(link_map_obj, reloc_index)
```

This function uses the link_map_obj parameter to access the information it needs to resolve the desired imported function (identified by the reloc_index argument) and writes the result into the appropriate GOT entry. After _dl_runtime_resolve resolves the imported function, control flow is passed to that function, making the resolution process completely transparent to the caller. The next time the PLT stub for the specified function is invoked execution will be diverted directly to the target function.

Listing 1: Example PLT and GOT.

```
100 PLT0:                  196 ; .plt.got start
100     push *0x200        196 ; Empty entry
106     jmp *0x204         196 0
110 printf@plt:            200 ; link_map object
110     jmp *0x208         200 &link_map_obj
116     push #0            204 ; Resolver function
11B     jmp PLT0           204 &_dl_runtime_resolve
120 read@plt:              208 ; printf entry
120     jmp *0x20C         208 0x116
126     push #1            20C ; read entry
12B     jmp PLT0           20C 0x126
```

The link_map structure contains all the information that the dynamic loader needs about a loaded ELF object. Each link_map instance is an entry in a doubly-linked list containing the information about all loaded ELF objects.

3.4 Symbol Versioning

The ELF standard provides a mechanism to import a symbol with a specific version associated with it. This feature is used to require a function to be imported from a

Table 1: Entries of the .dynamic section. d_tag is the key, while d_value is the value.

d_tag	d_value	d_tag	d_value
DT_SYMTAB	.dynsym	DT_PLTGOT	.got.plt
DT_STRTAB	.dynstr	DT_VERNEED	.gnu.version
DT_JMPREL	.rel.plt	DT_VERSYM	.gnu.version_r

specific version of a library. For instance, it is possible to require the fopen C Standard Library function, as implemented in version 2.2.5 of the GNU C Standard Library, using the version identifier GLIBC_2.2.5. The .gnu.version_r section contains version definitions in the form of Elf_Verdef structures.

The association between a dynamic symbol and the Elf_Verdef structure that it refers to is kept in the .gnu.version section, as an array of Elf_Verneed structures, one for each entry in the dynamic symbol table. These structures have a single field: a 16-bit integer that represents an index into the .gnu.version_r section.

Due to this layout, the index in the r_info field of the Elf_Rel structure is used by the dynamic loader as an index into both the .dynsym and .gnu.version sections. This is important to understand, as Leakless will later leverage this fact.

3.5 The .dynamic section and RELRO

The dynamic loader collects all the information that it needs about the ELF object from the .dynamic section, which is composed of Elf_Dyn structures. An Elf_Dyn is a key-value pair that stores different types of information. The relevant entries of this section, shown in Table 1, hold the absolute addresses of specific sections. One exception is the DT_DEBUG entry, which holds a pointer to an internal data structure of the dynamic loader. This is initialized by the dynamic loader and is used for debugging purposes.

An attacker able to tamper with these values can pose a security risk. For this reason, a protection mechanism known as RELRO (RELocation Read Only) has been introduced in dynamic loaders. RELRO comes in two flavors: partial and full.

Partial RELRO In this mode, some sections, including .dynamic, are marked as read-only after they have been initialized by the dynamic loader.

Full RELRO In addition to partial RELRO, lazy resolution is disabled: all import symbols are resolved at startup time, and the .got.plt section is completely initialized with the final addresses of the target functions and marked read-only. Moreover, since lazy resolution is not enabled, the GOT[1] and GOT[2] entries are not initialized with the values we mentioned in Section 3.3.

As we will see, RELRO poses significant complications that Leakless must (and does) address in order to operate in the presence of these countermeasures.

Note that the previously mentioned link_map structure stores in the l_info field an array of pointers to most of entries in the .dynamic section for internal usage. Since the dynamic loader trusts the content of this field implicitly, Leakless will later be able to misuse this to its own ends.

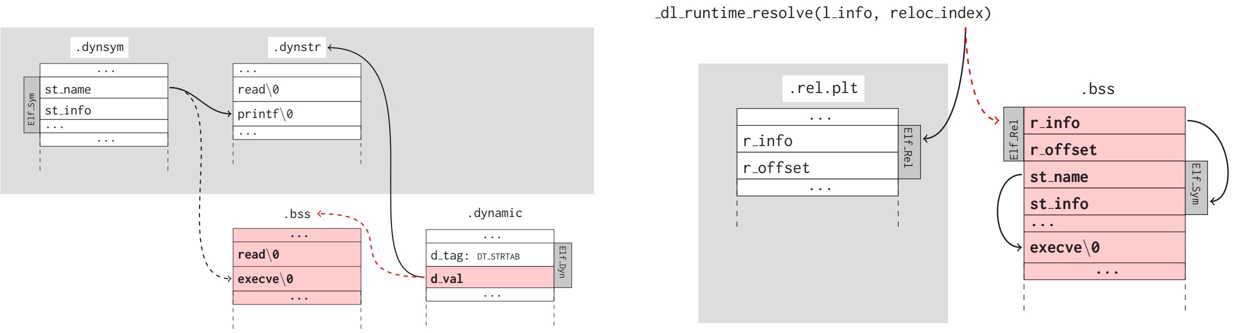
4 The Attack

Leakless enables an attacker to call arbitrary library functions, using only their name, without any information about the memory layout of the vulnerable program's libraries. To achieve this, Leakless abuses the dynamic loader, forcing it to resolve and call the requested function. This is possible for the same reason that memory corruption vulnerabilities are so damaging: the mixing of control data and non-control data in memory. In the case of a stack overflow, the control data in question is a stored return address. For the dynamic loader, the control data is comprised of the various data structures that the dynamic loader uses for symbol resolution. Specifically, the *name* of the function, stored in the .dynstr section, is analogous to a return address: it specifies a specific target to execute when the function is invoked.

The dynamic loader makes the assumption that the parameters it receives and its internal structures are trustworthy because it assumes that they are provided directly by the ELF file or by itself during initialization. However, when an attacker is able to modify this data, the assumption is broken. Some dynamic loaders (FreeBSD) validate the input they receive. However, they still implicitly trust the control structures, which will be readily corrupted by Leakless.

Leakless is designed to be used by an attacker who is attempting to exploit an existing vulnerability. The input to Leakless is comprised of the executable ELF file, a set of ROP gadgets of the binary (we detail what gadgets an attacker needs in Section 5.1), and the name of a library function that the attacker wishes to call (typically, but not necessarily, execve()). Given this information, Leakless outputs a ROP payload that executes the needed library function, bypassing various hardening techniques applied to the binary in question. This ROP chain is generally very short: depending on the mitigations present in the binary, the chain is 3 to 12 write operations. Some examples of the output produced by Leakless are available in the documentation of the Leakless code repository [17].

Leakless does *not* require any information about the addresses or contents of the libraries; we assume that ASLR is enabled for all dynamic libraries and that no knowledge about them is available. However we also assume that the executable is not position-independent, and, thus, is always loaded in a specific location in memory. We discuss this limitation in detail in Section 7.2, and show



(a) Example of the attack presented in Section 4.1. The attacker is able to overwrite the value of the DT_STRTAB dynamic entry, tricking the dynamic loader into thinking that the .dynstr section is in .bss, where he crafted a fake string table. When the dynamic loader will try to resolve the symbol for printf it will use a different base to reach the name of the function and will actually resolve (and call) execve.

(b) Example of the attack presented in Section 4.2. The reloc_index passed to _dl_runtime_resolve overflows the .rel.plt section and ends up in .bss, where the attacker crafted an Elf_Rel structure. The relocation points to an Elf_Sym located right afterwards overflowing the .dynsym section. In turn the symbol will contain an offset relative to .dynstr large enough to reach the memory area after the symbol, which contains the name of the function to invoke.

Figure 2: Illustration of some of the presented attacks. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker.

how infrequently Position Independent Executables (PIE) binaries occur in modern OS distributions in Section 6.2.

While in most cases, Leakless works independently of the dynamic loader implementation and version that the target system is running, some of our attacks require minor modifications to accommodate different dynamic loaders.

Note that Leakless's aim, obtaining the address of a library function and call it, is similar to what the dlsym function of libdl does. However, in practice this function is rarely used by applications and, therefore, its address is not generally known to the attacker.

4.1 The Base Case

As explained in Section 3 and illustrated in Figure 1, the dynamic loader starts its work from a Elf_Rel structure in the .rel.plt, then follows the index into the .dynsym section to locate the Elf_Sym structure, and finally uses that to identify the name (a string in the .dynstr section) of the symbol to resolve. The simplest way to call an arbitrary function would be to overwrite the string table entry of an existing symbol with the name of the desired function, and then invoke the dynamic loader, but this is not possible, as the section containing the string table for dynamic symbols, i.e., .dynstr, is not writeable.

However, the dynamic loader obtains the address of the .dynstr section from the DT_STRTAB entry of the .dynamic section, which is at a known location and, by default, writeable. Therefore, as shown in Figure 2a, it is possible to overwrite the d_val field of this dynamic entry with a pointer to a memory area under the control of the attacker (typically the .bss or .data section). This memory area would then include a single string, for example execve. At this point, the attacker needs to choose an existing symbol pointing to the correct offset in the fake string table and invoke the resolution of relocation corresponding to that symbol. This can be done by pushing the offset of this relocation on the stack and then jumping to PLT0.

This approach is simple, but it is only effective against binaries in which the .dynamic section is writeable. More sophisticated attacks must be used against binary compiled with partial or full RELRO.

4.2 Bypassing Partial RELRO

As we explained in Section 3.3, the second parameter of the _dl_runtime_resolve function is the offset of an Elf_Rel entry in the relocation table (.rel.plt section) that corresponds to the requested function. The dynamic loader takes this value and adds it to the base address of the .rel.plt to obtain the absolute address of the target Elf_Rel structure. However most dynamic loader implementations do not check the boundaries of the relocation table. This means that if a value larger than the size of the .rel.plt is passed to _dl_runtime_resolve, the loader will use the Elf_Rel at the specified location, despite being outside the .rel.plt section.

As shown in Figure 2b, Leakless computes an index that will lead _dl_runtime_resolve to look into a memory area under the control of the attacker. It then crafts an Elf_Rel structure that contains, in its r_offset field, the address of the writeable memory location where the address of the function will be written. The r_info field will, in turn, contain an index that causes the dynamic loader to look into the attacker-controlled memory. Leakless stores a crafted `Elf_Sym` object at this location, which, likewise, holds a `st_name` field value large enough to point into attacker-controlled memory. Finally, this location is where Leakless stores the name of the desired function to call.

In sum, Leakless crafts the full chain of structures involved in symbol resolution, co-opting the process to invoke the function whose name Leakless has written into attacker-controlled memory. After this, Leakless pushes the computed offset to the fake `Elf_Rel` structure onto the stack and invokes `PLT0`.

However, this approach is subject to several constraints. First, the symbol index in `Elf_Rel` has to be positive, since the `r_info` field is defined by the ELF standard as an unsigned integer. In practice, this means that the writable memory area (e.g., the `.bss` section) must be located *after* the `.dynsym` section. In our evaluation, this has always been the case.

Another constraint arises when the ELF makes use of the symbol versioning system described in Section 3.4. In this case, the `Elf_Rel.r_info` field is not just used as an index into the dynamic symbol table, but also as an index in the symbol version table (the `.gnu.version` section). In general, Leakless is able to automatically satisfy these constraints, except for x86-64 small binaries using huge pages [32]. We detail the additional constraints introduced by symbol versioning in Appendix A. When the constraints cannot be satisfied, an alternate approach must be adopted. This involves abusing the dynamic loader by corrupting its internal data structures to alter the dynamic resolution process.

4.3 Corrupting Dynamic Loader Data

We recall that the first parameter to `_dl_runtime_resolve` is a pointer to a data structure of type `link_map`. This structure contains information about the ELF executable, and the contents of this structure are implicitly trusted by the dynamic loader. Furthermore, Leakless can obtain the address of this structure from the second entry of the GOT of the vulnerable binary, whose location is deterministically known.

Recall from Section 3.5 that the `link_map` structure, in the `l_info` field, contains an array of pointers to the entries of the `.dynamic` section. These are the pointers that the dynamic loader uses to locate the objects that are used during symbol resolution. As shown in Figure 3, by overwriting part of this data structure, Leakless can make the `DT_STRTAB` entry of the `l_info` field point to a specially-crafted dynamic entry which, in turn, points to a fake dynamic string table. Hence, the attacker can reduce the situation back to the base case presented in Section 4.1.

This technique has wider applicability than the one presented in the previous section, since there are no specific constraints, and, in particular, it is applicable also against small 64 bit ELF binaries using huge pages. However, while in the previous attacks we were relying exclusively on standard ELF features, in this case (and in the one presented in the next section) we assume the layout of a glibc-specific structure (`link_map`) to be known. Each dynamic loader implements this structure in its own way, so minor modifications might be required when targeting a different dynamic loader. Note that `link_map`'s layout might change among versions of the same dynamic loader. However, they tend to be quite stable, and, in particular, in glibc no changes relevant to our attack have taken place since 2004.

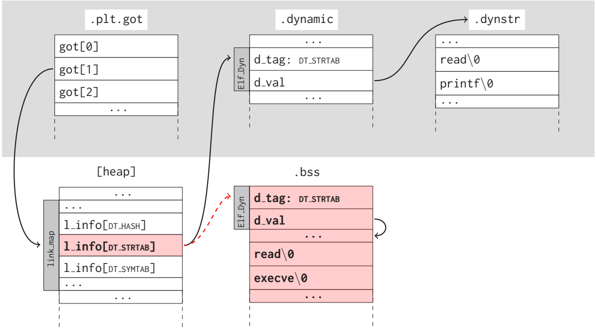


Figure 3: Example of the attack presented in Section 4.3. The attacker dereferences the second entry of the GOT and reaches the `link_map` structure. In this structure he corrupts the entry of the `l_info` field holding a pointer to the `DT_STRTAB` entry in the dynamic table. Its value is set to the address of a fake dynamic entry which, in turn, points to a fake dynamic string table in the `.bss` section.

4.4 The Full RELRO Situation

Leakless is able to bypass full RELRO protection.

When full RELRO is applied, all the relocations are resolved at load-time, no lazy resolving takes place, and the addresses of the `link_map` structure and of `_dl_runtime_resolve` in the GOT are never initialized. Thus, it is not directly possible to know their addresses, which is what the general technique to bypass partial RELRO relies upon.

However, it is possible to indirectly recover these two values through the `DT_DEBUG` entry in the dynamic table. The value of the `DT_DEBUG` entry is set by the dynamic loader at load-time to point to a data structure of type `r_debug`. This data structure contains information used by debuggers to identify the base address of the dynamic loader and to intercept certain events related to dynamic loading. In addition, the `r_map` field of this structure holds a pointer to the head of the linked list of `link_map` structures.

Leakless corrupts the first entry of the list describing the ELF executable so that the `l_info` entry for `DT_STRTAB` points to a fake dynamic string table. This is presented in

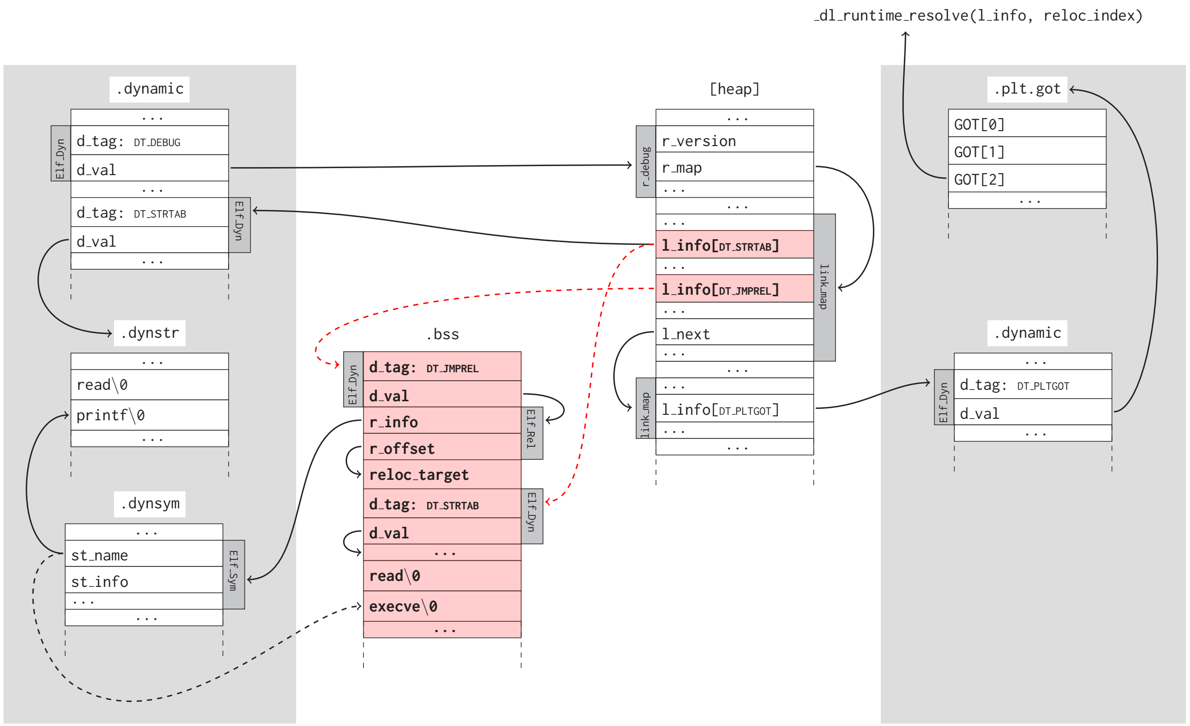


Figure 4: Example of the attack presented in Section 4.4. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker. The attacker goes through the DT_DEBUG dynamic entry to reach the r_debug structure, then, dereferencing the r_map field, he gets to the link_map structure of the main executable, and corrupts l_info[DT_STRTAB] as already seen in Section 3.
Since the .got.plt section is read-only due to full RELRO, the attacker also have to forge a relocation. To do so, he corrupts l_info[DT_JMPREL] making it point to a fake dynamic entry in turn pointing to a relocation. This relocation refers to the existing printf symbol, but has an r_offset pointing to a writeable memory area.
Then the attacker also needs to recover the pointer to the _dl_runtime_resolve function, which is not available in the GOT of the main executable due to full RELRO, therefore he dereferences the l_info field of the first link_map structure and gets to the one describing the first shared library, which is not protected by full RELRO. The attacker accesses the l_info[DT_PLTGOT] field and gets to the corresponding dynamic entry (the .dynamic on the right), and then to the .plt.got section (always on the right), at the second entry of which he can find the address of _dl_runtime_resolve.

Figure 4.

After this, Leakless must invoke _dl_runtime_resolve, passing the link_map structure that it just corrupted as the first argument and an offset into the new .dynsym as the second parameter. However, as previously mentioned, _dl_runtime_resolve is not available in the GOT due to full RELRO. Therefore, Leakless must look for its address in the GOT of *another* ELF object, namely, a library loaded by the application that is not protected by full RELRO. In most cases, only ELF executables are compiled with full RELRO, and libraries are not. This is due to the fact that RELRO is designed to harden, at the cost of performance, specific applications that are deemed “risky”. Applying full RELRO to a shared library would impact the performance of all applications making use of this library, and thus, libraries are generally left unprotected. Since the order of libraries in the linked list is deterministic, Leakless can dereference the l_next entry in link_map to reach the entry describing a library that is not protected by full RELRO, dereference the entry in l_info corresponding to the DT_PLTGOT dynamic entry, dereference its value (i.e., the base address of that library’s GOT), and read the address of _dl_runtime_resolve from this GOT.

Leakless must then overcome a final issue: _dl_runtime_resolve will not only call the target function, but will also try to write its address to the appropriate GOT entry. If this happens, the program will crash, as the GOT is read-only when full RELRO is applied. We can circumvent this issue by faking the DT_JMPREL dynamic entry in the link_map structure that points to the .rel.dyn section. Leakless points it to an attacker-controlled memory area and writes an

Elf_Rel structure, with a target (r_offset field) pointing to a writeable memory area, referring to the symbol we are targeting. Therefore, when the library is resolved, the address will be written to a writeable location, the program will not crash, and the requested function will be executed.

5 Implementation

Leakless analyzes a provided binary to identify which of its techniques is applicable, crafts the necessary data structures, and generates a ROP chain that implements the chosen technique. The discovery of the initial vulnerability itself, and the automatic extraction of usable gadgets from a binary are orthogonal to the scope of our work, and have been well-studied in the literature and implemented in the real world [6, 16, 19, 20, 34, 38]. We designed Leakless to be compatible with a number of gadget finding techniques, and have implemented a manual backend (where gadgets are provided by the user) and a backend that utilizes ROPC [22], an automated ROP compiler prototype built on the approach proposed by Q [34].

We also developed a small test suite, composed of a small C program with a stack-based buffer overflow compiled, alternatively, with no protections, partial RELRO, and full RELRO. The test suite runs on GNU/Linux with the x86, x86-64 and ARM architectures and with FreeBSD x86-64.

5.1 Required Gadgets

Leakless comprises four different techniques that are used depending on the hardening techniques applied to the binary. These different techniques require different gadgets to be provided to Leakless. A summary of the types of gadgets is presented in Table 2. The write_memory gadget is mainly used to craft data structures at known memory locations, while the deref_write gadget to traverse and corrupt data structures (in particular link_map). The deref_save and copy_to_stack gadgets are used only in the full RELRO case. The aim of the former is to save at a known location the address of link_map and _dl_runtime_resolve, while the latter is used to copy link_map and the relocation index on the stack before calling _dl_runtime_resolve, since using PLT0 is not a viable solution.

For the interested reader, we provide in-depth examples of executions of Leakless in the presence of two different sets of mitigation techniques in the documentation of the Leakless code repository [17].

6 Evaluation

We evaluated Leakless in four ways. First, we determined the applicability of our technique against different dynamic loader implementations. We then analyzed the binaries distributed by several popular GNU/Linux and BSD distributions (specifically, Ubuntu, Debian, Fedora, and FreeBSD) to determine the percentage of binaries that would be susceptible to our attack. Then we applied Leakless in two real-world exploits against a vulnerable version of Wireshark and in a more sophisticated attack against Pidgin. Finally we used a Turing-complete ROP compiler to implement the approach used in Leakless and two other previously used techniques, and compared the size of the resulting chains.

6.1 Dynamic Loaders

To show Leakless' generality, especially across different ELF-based platforms, we surveyed several implementations of dynamic loaders. In particular, we found that the dynamic loader part of the *GNU C Standard Library* (also known as glibc and widely used in GNU/Linux distributions), several other Linux implementations such as *dietlibc*, *uClibc* and *newlib* (widespread in embedded systems) and the *OpenBSD* and *NetBSD* implementations are vulnerable to Leakless. Another embedded library, *musl*, instead, is not susceptible to our approach since it does not support lazy loading. *Bionic*, the C Standard Library used in Android, is also not vulnerable since it only supports PIE binaries. The most interesting case, out of all the loaders we analyzed, is *FreeBSD*'s implementation. In fact, it is the only one which performs boundary checks on arguments passed to _dl_runtime_resolve. All other loaders implicitly trust input arguments argument. Furthermore, *all* analyzed loaders implicitly trust the control structures that Leakless corrupts in the course of most of its attacks.

In summary, out of all of the loaders we analyzed, only two are immune to Leakless by design: musl, which does not support lazy symbol resolution, and bionic, which only supports PIE executables. Additionally, because the FreeBSD dynamic loader performs bounds checking, the technique explained in Section 4.2 is not applicable. However, the other techniques still work.

6.2 Operating System Survey

To understand Leakless' impact on real-world systems, we performed a survey of all binaries installed in default installations of several different Linux and BSD distributions. Specifically, we checked all binaries in /sbin, /bin, /usr/sbin, and /usr/bin on these systems and classified the binaries by the applicability of the techniques used by Leakless. The distributions that we considered were Ubuntu 14.10, Debian Wheezy, Fedora 20, and FreeBSD 10. We used both x86 and x86-64 versions of these systems. On Ubuntu and Debian, we additionally installed the LAMP (Linux, Apache, MySQL, PHP) stack as an attempt to simulate a typical server deployment configuration.

The five categories that we based our ratings on are as

Table 2: Gadgets required for the various approaches. The “Signature” column represents the name of the gadget and the parameters it accepts, while “Implementation” presents the behavior of the gadget in C-like pseudo code. The last four columns indicate whether a certain gadget is required for the corresponding approach presented in Section 4. Under RELRO, “N” indicates RELRO is disabled, “P” means partial RELRO is used, “H” stands for the partial RELRO and small 64 bit binaries using huge pages, and “F” denotes that full RELRO is enabled.

		RELRO			
Signature	Implementation	N	P	H	F
write_memory $(destination, value)$	$*(destination) = value$	✓	✓	✓	✓
deref_write$(pointer, offset, value)$	$*(*(pointer)+offset) = value$			✓	✓
deref_save $(destination, pointer, offset)$	$*(destination) = *(*(pointer)+offset)$				✓
copy_to_stack $(offset, source)$	$*(stack\_pointer+offset) = *(source)$				✓

follows:

Unprotected. This category includes binaries that have no RELRO or PIE. For these binaries, Leakless can apply its base case technique, explained in Section 4.1.

Partial RELRO. Binaries that have partial RELRO, but lack PIE, fall into this category. In this case, Leakless would apply the technique described in Section 4.2.

Partial RELRO (huge pages). Binaries in this category have partial RELRO, use huge pages, and are very small, therefore, they require Leakless to use the technique described in Section 4.3. They are included in this category.

Full RELRO. To attack binaries that use full RELRO, which comprise this category, Leakless must apply the technique presented in Section 4.4.

Not susceptible. Finally, we consider binaries that use PIE to be insusceptible to Leakless (further discussion on this in Section 7.2).

The results of the survey, normalized to the total number of binaries in an installation, are presented in Figure 5. We determined that, on Ubuntu, 84% of the binaries were susceptible to at least one of our techniques and 16% were protected with PIE. On Debian, Leakless can be used on 86% of the binaries. Fedora has 76% of susceptible binaries. Interestingly, FreeBSD ships no binaries with RELRO or PIE, and is thus 100% susceptible to Leakless.

Additionally, we performed a survey on the shared libraries of the systems we considered. We found that, on average, only 11% of the libraries had full RELRO protection. This has some interesting implications for Leakless: for a given binary, the likelihood of finding a loaded library without full RELRO is extremely high and, even if a vulnerable binary employs RELRO, Leakless can still apply its full RELRO attack to bypass this. This has the effect of making RELRO basically useless as a mitigation technique, unless it is applied system-wide.

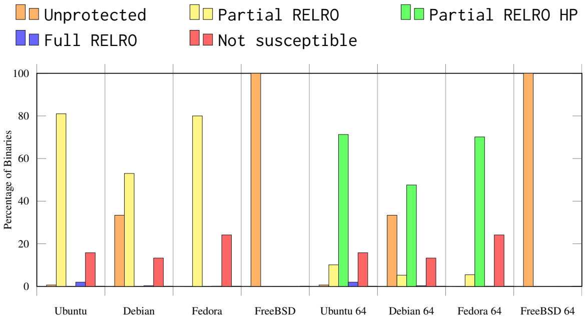


Figure 5: Classification of the binaries in default installations of target distributions. Binaries marked as `Unprotected`, `Partial RELRO`, `Partial RELRO HP` and `Full RELRO` require, respectively, to the attacks detailed in Sections 4.1, 4.2, 4.3 and 4.4, while for `Not susceptible` binaries, the Leakless approach is not applicable

6.3 Case Study: Wireshark

We carried out a case study in applying Leakless to a vulnerability in a program that does not present a direct line of communication to the attacker. In other words, the exploit must be done in one-shot, with no knowledge of the layout of the address space or the contents of libraries.

We picked a recent (April 2014) vulnerability [7], which is a stack-based buffer overflow in *Wireshark*’s MPEG protocol parser in versions 1.8.0 through 1.8.13 and 1.10.0 through 1.10.6. We carried out our experiments against a Wireshark 1.8.2 binary compiled with partial RELRO and one compiled with full RELRO. Both were compiled for x86-64 on Debian Wheezy and used the GNU C Library, without other protections such as PIE and stack canaries.

We used the manual Leakless backend to identify the required gadgets to construct the four necessary primitives (described in Section 5.1): *write_memory*, *deref_write*, *deref_save* and *copy_to_stack*. In the case of Wireshark, it was trivial to find gadgets to satisfy all of these primitives.

Leakless was able to construct a one-shot exploit using the attacks presented in Section 4.2 and Section 4.4. In both cases, the exploit leverages the dynamic loader in order to call the `execve` function from glibc to launch an executable of our choice.

6.4 Case Study: Pidgin

We also applied Leakless to Pidgin, a popular multi-protocol instant-messaging client, to build a more sophisticated exploit. Specifically, we wanted to perform a malicious operation without calling any anomalous system call (e.g. `execve("/bin/sh")`) which could trigger intrusion detection systems. We used Pidgin 2.10.7, building it from the official sources with RELRO enabled and targeting the x86 architecture.

To this end, we crafted an exploit designed to masquerade itself in legitimate functionality present in the application logic: tunneling connections through a proxy. The idea of the attack is that an IM service provider exploits a vulnerability such as CVE-2013-6487 [14] to gain code execution, and, using Pidgin's global proxy settings, redirects all IM traffic through a third-party server to enable chat interception.

Once we identified the necessary gadgets to use Leakless with full RELRO protection, it was easy to invoke functions contained in `libpurple.so` (where the core of the application logic resides) to perform the equivalent of the C code shown in Listing 2.

Listing 2: The Pidgin attack.

```
void *p, *a;
p = purple_proxy_get_setup(0);
purple_proxy_info_set_host(p, "legit.com");
purple_proxy_info_set_port(p, 8080);
purple_proxy_info_set_type(p, PURPLE_PROXY_HTTP);

a = purple_accounts_find("usr@xmpp", "prpl-xmpp");
purple_account_disconnect(a);
purple_account_connect(a);
```

Interestingly, some of this library-provided functionality is not imported into the Pidgin executable itself, and would be very challenging to accomplish in a single-stage payload, without Leakless.

6.5 ROP chain size comparison

To prove the effectiveness of the Leakless approach, we compared it with two existing techniques that allow an attacker to call arbitrary library functions. The first consists in scanning a library backwards, starting from an address in the `.plt.got` section, until the ELF header is found, and then scan forward to find a fingerprint of the function the attacker wants to invoke. This approach is feasible, but not very reliable, since different versions (or implementations) of a library might not be uniquely identified with a single fingerprint. The second technique is more reliable, since it implements the full symbol resolution process, as it is carried out by the dynamic loader.

Table 3: Size of the ROP chains generated by ROPC for each technique presented in Section 6.5, and by Leakless' manual backend (*). The second column represents the size in bytes for the setup and the first call, while the third column shows the additional cost (in bytes) for each subsequent call. Finally, the fourth column indicates the percentage of vulnerabilities used in Metasploit that would be feasible to exploit with a ROP chain of the *First call* size.

Technique	First call	Subsequent	Feasibility
ROPC - scan library	3468 bytes	+340 bytes	16.38%
ROPC - symbol resolution	7964 bytes	+580 bytes	8.67%
Leakless partial RELRO	648 bytes	+84 bytes	73.78%
Leakless full RELRO	2876 bytes	+84 bytes	17.44%
Leakless* partial RELRO	292 bytes	+48 bytes	95.24%
Leakless* full RELRO	448 bytes	+48 bytes	88.9%

We implemented these two approaches using a Turing-complete ROP compiler for x86, based on Q [34], called ROPC [22]. We compare these approaches against that of Leakless' ROPC backend, in partial RELRO and full RELRO modes. For completeness, we also include the Leakless' manual backend, with gadgets specified by the user.

In fact, the size of a ROP payload is critical, vulnerabilities often involve an implicit limit on the size of the payload that can be injected into a program. To measure the impact of Leakless' ROP chain size, we collected the size limits imposed on payloads of all the vulnerability descriptions included in the *Metasploit Framework* [31], a turn-key solution for launching exploits against known vulnerabilities in various software. We found that 946 of the 1,303 vulnerability specifications included a maximum payload size, with an average specified maximum payload size of 1332 bytes. To demonstrate the increase in the feasibility of automatically generating complex exploits, we include, for each evaluated technique, the percentage of Metasploit vulnerabilities for which the technique can automatically produce short enough ROP chains.

The results, in terms of length of the ROP chain generated for ROPC's test binaries and feasibility against the vulnerabilities used in Metasploit, are shown in Table 3. Leakless outperforms existing techniques, not only in the absolute size of the ROP chain to perform the initial call, but also in the cost of performing each additional call, which is useful in a sophisticated attack such as the one demonstrated in Section 6.4.

7 Discussion

In this section, we will discuss several aspects relating to Leakless: why the capabilities that it provides to attackers are valuable, when it is most applicable, what its limitations are, and what can be done to mitigate against them.

7.1 Leakless Applications

Leakless represents a powerful tool in the arsenal of exploit developers, aiding them in three main areas: functionality reuse, one-shot exploitation, and ROP chains shortening.

One-shot exploitation. While almost any exploit can be simplified by Leakless, we have designed it with the goal of enabling exploits that, without it, require an information disclosure vulnerability, but for which an information disclosure is not feasible or desirable. A large class of programs that fall under this category are file format parsers.

Code that parses file formats is extremely complex and, due to the complex, untrusted input that is involved, this code is prone to memory corruption vulnerabilities. There are many examples of this: the image parsing library `libpng` had 27 CVE entries over the last decade [10], and `libtiff` had 53 [11]. Parsers of complex formats suffer even more: the multimedia library `ffmpeg` has accumulated 170 CVE entries over the last five years alone [9]. This class of libraries is not limited to multimedia. Wireshark, a network packet analyzer, has 285 CVE entries, most of which are vulnerabilities in network protocol analysis plugins [12].

These libraries, and others like them, are often used *offline*. The user might first download a media or PCAP file, and *then* parse it with the library. At the point where the vulnerability triggers, an attacker cannot count on having a direct connection to the victim to receive an information disclosure and send additional payloads. Furthermore, most of these formats are *passive*, meaning that (unlike, say, PDF), they cannot include scripts that the attacker can use to simulate a two-step exploitation. As a result, even though these libraries might be vulnerable, exploits for them are either extremely complex, unreliable, or completely infeasible. By avoiding the information disclosure step, Leakless makes these exploits simpler, reliable, and feasible.

Functionality reuse. Leakless empowers attackers to call arbitrary functions from libraries loaded by the vulnerable application. In fact, the vulnerable application does not have to actually *import* this function; it just needs to link against the library (i.e., call any other function in the library). This is brings several benefits.

To begin with, the C Standard Library, which is linked against by most applications, includes functions that wrap almost every system call (e.g., `read()`, `execve()`, and so on). This means that Leakless can be used to perform any system call, in a short ROP chain, even without a system call gadget.

Moreover, as demonstrated in Section 6.4, Leakless enables easy reuse of existing functionality present in the application logic. This is important for two reasons.

First, this helps an attacker perform stealthy attacks by making it easier to masquerade an exploit as something the application might normally do. This can be crucial when a standard exploitation path is made infeasible by the presence of protection mechanisms such as seccomp [2], AppArmor [1], or SELinux [25].

Second, depending on the goals of the attacker, reusing program functionality may be better than simply executing arbitrary commands. Aside from the attack discussed in our Pidgin case study, an attacker can, for example, silently enable insecure cipher-suites, or versions of SSL, in the Firefox web browser with a single function call to `SSL_CipherPrefSetDefault` [24].

Shorter ROP chains. As demonstrated in Section 6.5, Leakless produces shorter ROP chains than existing techniques. In fact, in many cases, Leakless is able to produce ROP chains less than one kilobyte that lead to the execution of arbitrary functions. As many vulnerabilities have a limit as to the maximum size of the input that they will accept, this is an important result. For example, the vulnerability that we exploited in our Pidgin case study allowed a maximum ROP chain of one kilobyte. Whereas normal ROP compilation techniques would be unable to create automatic payloads for this vulnerability, Leakless was able to call arbitrary functions via an automatically-produced ROP chain that remained within the length limit.

7.2 Limitations

Leakless' biggest limitation is the inability to handle Position Independent Executables (PIEs) without a prior information disclosure. This is a general problem to any technique that uses ROP, as the absolute addresses of gadgets must be provided in the ROP chain. Additionally, without the base address of the binary, Leakless would be unable to locate the dynamic loader structures that it needs to corrupt.

When presented with a PIE executable, Leakless requires the attacker to provide the application's base address, which is presumably acquired via an information disclosure vulnerability (or, for example, by applying the technique presented in BROP [5]). While this breaks Leakless' ability to operate without an information disclosure, Leakless is likely still the most convenient way to achieve exploitation, as no library locations or library contents have to be leaked. Additionally, depending on the situation, the disclosure of just the address of the binary might be more feasible than the disclosure of the *contents* of an entire library. Unlike other techniques, which may need the latter, Leakless only requires the former.

In practice, PIEs are uncommon due to the associated cost in terms of performance. Specifically, measurements have shown that PIE overhead on x86 processors averages at 10%, while the overhead on x86-64 processors, thanks to instruction-pointer-relative addressing, averages at 3.6% [28].

Because of the overhead associated with PIE, most distributions ship with PIE enabled only for those applications deemed “risky”. For example, according to their documentation, Ubuntu ships only 27 of their officially supported packages (i.e., packages in the “main” repository) with PIE enabled, out of over 27,000 packages [40]. As shown in Section 6.1, PIE executables comprise a minority of the executables on all of the systems that we surveyed.

7.3 Countermeasures

There are several measures that can be taken against Leakless, but they all have drawbacks. In this sections we analyze the most relevant ones.

Position Independent Executables. A quick countermeasure is to make every executable on the system position independent. While this would block Leakless’s automatic operation (as discussed in Section 7.2), it would still allow the application of the Leakless technique when any information disclosure does occur. For that reason, and the performance overhead associated with PIE, we consider the other countermeasures described in this section to be better solutions to the problem.

Disabling lazy loading. When the `LD_BIND_NOW` environment variable is set, the dynamic loader will completely disable *lazy loading*. That is, all imports, for the program binary and any library it depends on, are resolved upon program startup. As a side-effect of this, the address of `_dl_runtime_resolve` does not get loaded into the GOT of any library, and Leakless cannot function. This is equivalent to enable full RELRO on the whole system, and consequently, it incurs in the same, non-negligible, performance overhead.

Disabling DT_DEBUG. Finally, Leakless also uses the `DT_DEBUG` dynamic entry, used by debuggers for intercepting loading-related events, to bypass full RELRO. Currently, this field is always initialized, opening the doors for Leakless’ full RELRO bypass. To close this hole, the dynamic loader could be modified to only initialize this value when a debugger is present or in the presence of an explicitly-set environment variable.

Better protection of loader control structures. Leakless heavily relies on the fact that dynamic loader control structures are easily accessible in memory, and their locations are well-known. It would be beneficial for these structures to be better protected, or hidden in memory, instead of being loaded at a known location. For example, as shown in [29], these structures, along with any sections that provide control data for symbol resolution, could be marked as read-only after initialization. Such a development would eliminate Leakless’ ability to corrupt these structures and would prevent the attack from redirecting the control flow to sensitive functions.

Additionally, modifying the loading procedure to use a table of `link_map` structure, and letting `_dl_runtime_resolve` take an index in this table, instead of a direct pointer, will break Leakless’ bypass of full RELRO. However, this change would also break compatibility with any binaries compiled before the change is implemented.

Isolation of the dynamic loader. Isolating the dynamic loader from the address space of the target program could be an effective countermeasure. For instance, on Nokia’s *Symbian OS*, which has a micro-kernel, the dynamic loader is implemented in a separate process as a *system server* which interfaces with the kernel [26]. This guarantees that the control structures of the dynamic loader cannot be corrupted by the program, and, therefore, this makes Leakless virtually ineffective. However, such a countermeasure would have a considerable impact on the overall performance of applications due to the overhead of IPC (Inter-Process Communication).

In general, the mitigations either represent a runtime performance overhead (PIE or loader isolation), a load-time performance overhead (non-lazy loading and system-wide RELRO), or a modification of the loading process (`DT_DEBUG` disabling or loader control structure hiding). In the long run, we believe that a redesign of the dynamic loader, with security in mind, would be extremely beneficial to the community. In the short term, there are options available to protect against Leakless, but they all come with a performance cost.

8 Conclusion

In this paper, we presented Leakless, a new technique that leverages functionality provided by the dynamic loader to enable attackers to use arbitrary, security-critical library functions in their exploits, without having to know where in the application’s memory these functions are located. This capability allows exploits that, previously, required an information disclosure step to function.

Since Leakless leverages features mandated in the ELF binary format specification, the attacks it implements are applicable across architectures, operating systems, and dynamic loader implementations. Additionally, we showed how our technique can be used to bypass hardening schemes such as RELRO, which are designed to protect important control structures used in the dynamic resolution process. Finally, we proposed several countermeasures against Leakless, discussing the advantages and disadvantages of each one.

References

[1] AppArmor. http://wiki.apparmor.net/.

[2] A. Arcangeli. seccomp. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[3] A. Baratloo, N. Singh, and T. K. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.

[4] M. Bauer. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 2006(148):13, 2006.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[7] Common Vulnerabilities and Exposures. CVE-2014-2299. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299.

[8] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[9] CVEDetails.com. ffmpeg: CVE security vulnerabilities. http://www.cvedetails.com/product/6315/Ffmpeg-Ffmpeg.html.

[10] CVEDetails.com. Libpng: Security Vulnerabilities. http://www.cvedetails.com/vendor/7294/Libpng.html.

[11] CVEDetails.com. Libtiff: CVE security vulnerabilities. http://www.cvedetails.com/product/3881/Libtiff-Libtiff.html.

[12] CVEDetails.com. Wireshark: CVE security vulnerabilities. http://www.cvedetails.com/product/8292/Wireshark-Wireshark.html.

[13] CWE. CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[14] N. V. Database. NVD - Detail - CVE-2013-6487. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6487.

[15] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Leakless source code repository. https://github.com/ucsb-seclab/leakless.

[16] S. Dudek. The Art Of ELF: Analysis and Exploitations. http://bit.ly/1a8MeEw.

[17] T. Dullien, T. Kornau, and R.-P. Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In *WOOT*, 2010.

[18] M. Fox, J. Giordano, L. Stotler, and A. Thomas. Selinux and grsecurity: A case study comparing linux security kernel enhancements. 2009.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security*, pages 49–64, 2013.

[20] C. Heitman and I. Arce. BARFgadgets. https://github.com/programa-stic/barf-project/tree/master/barf/tools/gadgets.

[21] inaz2. ROP Illmatic: Exploring Universal ROP on glibc x86-64. http://ja.avtokyo.org/avtokyo2014/speakers#inaz2.

[22] P. Kot. A Turing complete ROP compiler. https://github.com/pakt/ropc.

[23] P. Menage. Cgroups. *Available on-line at: http://www. mjmwired. net/kernel/Documentation/c-groups. txt*, 2008.

[24] Mozilla. SSL_CipherPrefSetDefault. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#__SSL_CipherPrefSetDefault_.

[25] National Security Agency. Security-Enhanced Linux. http://selinuxproject.org/.

[26] Nokia. Symbian OS Internals - The Loader. http://developer.nokia.com/community/wiki/Symbian_OS_Internals/10._The_Loader#The_loader_server.

[27] H. Orman. The Morris worm: a fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[28] M. Payer. Too much PIE is bad for performance. 2012. https://nebelwelt.net/publications/12TRpie/gccPIE-TR120614.pdf.

[29] M. Payer, T. Hartmann, and T. R. Gross. Safe Loading - A Foundation for Secure Execution of Untrusted Programs. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 18–32, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Phrack. Phrack - Volume 0xB, Issue 0x3a. `http://phrack.org/issues/58/4.html`.

[31] Rapid7, Inc. The Metasploit Framework. `http://www.metasploit.com/`.

[32] RedHat, Inc. Huge Pages and Transparent Huge Pages. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html`.

[33] Santa Cruz Operation. System V Application Binary Interface, 2013. `http://www.sco.com/developers/gabi/latest/contents.html`.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, 2011.

[35] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[36] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 11–11, Berkeley, CA, USA, 2013. USENIX Association.

[37] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[38] The Avalanche Project. Avalange - a dynamic defect detection tool. `https://code.google.com/p/avalanche/`.

[39] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[40] Ubuntu. Ubuntu Wiki - Security/Features. `https://wiki.ubuntu.com/Security/Features#Built_as_PIE`.

[41] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security Symposium*, pages 29–46, 2010.

A Symbol versioning challenges

In Section 3.4 we introduced the concept of symbol versioning, and in Section 4.2 we mentioned that its usage introduces additional constraints in the value that `Elf_Rel.r_info` can assume. In this Appendix we illustrate these constraints, and how Leakless can automatically verify and satisfy them.

A.1 Constraints due to symbol versioning

In presence of symbol versioning, the `Elf_Rel.r_info` field is used both as an index into the dynamic symbol table and as an index in the symbol version table (the `.gnu.version` section), which is composed by `Elf_Verneed` values. An `Elf_Verneed` value of zero or one has a special meaning, and stops the processing of the symbol version, which is a desirable situation for the attacker.

To understand the constraints posed by this, we introduce some definitions and naming conventions. *idx* is the index in `Elf_Rel.r_info` that Leakless has computed, baseof(x) is the function returning the base address of section x, sizeof(y) is the function returning the size in bytes of structure y, and $*$ is the pointer dereference operator. We define the following variables:

$$\begin{aligned}
sym &= \text{baseof}(\texttt{.dynsym}) + idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) \\
ver &= \text{baseof}(\texttt{.gnu.version}) + \\
&\quad + idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) \\
verdef &= \text{baseof}(\texttt{.gnu.version\_r}) + \\
&\quad + *(ver) \cdot \text{sizeof}(\texttt{Elf\_Verdef})
\end{aligned}$$

To be able to carry on the attack, the following conditions must hold:

1. *sym* points to a memory area controlled by the attacker, and
2. one of the following holds:
   (a) *ver* points to a memory area containing a zero or a one, or
   (b) *ver* points to a memory area controlled by the attacker, which will write a zero value there, or
   (c) *verdef* points to a memory area controlled by the attacker, which will place there an appropriately crafted `Elf_Verdef` structure.

All the other options result in an access to an unmapped memory area or the failure of the symbol resolution process, both of which result in program termination.

Leakless is able to satisfy these constraints automatically in most cases. The typical successful situation results in an *idx* value that points to a version index with value zero or one in the `.text` section (which usually comes after `.gnu.version`) and to a symbol in the `.data` or `.bss` section. A notable exception, where this is impossible to achieve, is in the case of small x86-64 ELF binaries compiled with the support of huge pages [32]. Using huge pages means that memory pages are aligned to boundaries of 2 MiB and, therefore, the segment containing the read-only sections (in particular, `.gnu.version` and `.text`) is quite far from the writeable segment (containing `.bss` and `.data`). This makes it hard to find a good value for *idx*.

A.2 The huge page issue

The effect of huge pages can be seen in the following examples:

```
$ readelf --wide -l elf-without-huge-pages

Program Headers:
  Type VirtAddr   MemSiz   Flg Align
  ...
  LOAD 0x00400000 0x006468 R E 0x1000
  LOAD 0x00407480 0x0005d0 RW  0x1000
  ...

$ readelf --wide -l elf-with-huge-pages

Program Headers:
  Type VirtAddr   MemSiz   Flg Align
  ...
  LOAD 0x00400000 0x00610c R E 0x200000
  LOAD 0x00606e10 0x0005d0 RW  0x200000
  ...
```

While in the first case the distance between the beginning of the executable and the writeable segments is in the order of the kilobytes, with huge pages is more than 2 MiB, and a valid value for *idx* cannot be found.

There are two ways to resolve the problems posed to Leakless by small 64-bit binaries.

The first option is to find a zero value for `Elf_Verneed` in the read-only segment (usually `.text`). Given *ro_start*, *ro_end* and *ro_size*, as the start and end virtual addresses and the size of the read-only segment respectively, and *rw_start*, *rw_end* and *rw_size* as the respective values for the writeable segment, the following must hold:

$$\begin{aligned} ro\_start &\leq \ ver \ < ro\_end \\ rw\_start &\leq sym < rw\_end \end{aligned}$$

Here, the most difficult case to satisfy is if `.dynsym` or `.gnu.version` start at *ro_start*. If we assume that *both* hold true, we can write the following:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &< ro\_end - ro\_start \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &\geq rw\_start - ro\_start \end{aligned}$$

Or, alternatively:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &< ro\_size \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &\geq 2 \text{ MiB} \end{aligned}$$

Knowing that `Elf_Verneed` and `Elf_Sym` have, respectively, a size of 2 and 24 bytes for 64 bit ELFs, we can compute the minimum value of `ro_size` to make this system of inequalities satisfiable. The result is 170.7 KiB. If the `.rodata` section is smaller than this size, an alternative method must be used.

The second option is to position `Elf_Verneed` in the writeable segment. In this case, the attack requirements can be described by the following system of inequalities:

$$\begin{aligned} rw\_start &\leq \ ver \ < rw\_end \\ rw\_start &\leq sym < rw\_end \end{aligned}$$

If we, once again, consider the most stringent constraints and apply the previously mentioned assumptions, we get the following:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &\geq rw\_start - ro\_start \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &< rw\_start - ro\_start + \\ &\quad + rw\_size \end{aligned}$$

Or, alternatively:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &\geq 2 \text{ MiB} \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &< 2 \text{ MiB} + rw\_size \end{aligned}$$

We can now put a lower bound on the size of the writeable segment (*rw_size*) to make the system satisfiable: 22 MiB. However, this is unreasonably large, and leads us to the conclusion that this approach is not viable with small 64 bit ELF binaries that use huge pages.