"Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata

Rebecca Shapiro
Dartmouth College

Sergey Bratus
Dartmouth College

Sean W. Smith
Dartmouth College

Abstract

Although software exploitation historically started as an exercise in coaxing the target's execution into attacker-supplied binary shellcode, it soon became a practical study in pushing the limits of unexpected computation that could be caused by crafted data not containing any native code. We show how the ABI metadata that drives the creation of a process' runtime can also drive arbitrary computation. We introduce our design and implementation of Cobbler, a proof-of-concept toolkit capable of compiling a Turing-complete language into well-formed ELF executable metadata that get "executed" by the runtime loader (RTLD). Our proof-of-concept toolkit highlights how important it is that defenders expand their focus beyond the code and data sections of untrusted binaries, both in static analysis and in the dynamic analysis of the early runtime setup stages as well as any time the RTLD is invoked.

1 Introduction

The great threat model change. This last decade saw a major change in the underlying threat model of applied security research: the change from the *"malicious code" model* (where the attacker slips a native code payload/shellcode into the target system and tricks the target into passing control into it) to *"malicious computation"* driven merely by attacker-crafted *data* with no native code payloads (see, e.g., discussion in [23]). This new attack model not only highlighted the ineffectiveness of defensive mechanisms that attempted to detect and block "malicious code" in system's inputs or communications, but also helped clarify the core essence of exploitation: unexpected computation in the target, caused by crafted inputs.[1]

1.1 Data-driven attacks

The ability of non-code crafted inputs to drive "programs" (chains of the target's own code fragments, function calls, or even dynamic linker invocations) in the target has been discussed by hacker researchers since at least the early 2000s (e.g., in [37, 32], see also [26, 15, 8] for historical analysis of these discoveries); other researchers also pointed have out the power of data-only attacks to alter target's execution [11]. For any sufficiently complex data, the environment that processes these data may play the role of a "virtual machine" programmed by such data acting as "byte-code" unless we can demonstrate that such computation is limited by design [6, 35]. Typically these data that drive these "weird machines" are either involved in program flow control or in memory transformations. For example, stack frames can drive return-oriented programming (ROP); exception handling data used for stack unwinding and state recovery can drive DWARF-based illicit computation [33]; and heap chunk metadata can be used to manipulate memory management code [4, 29, 24, 22, 36] (also see [5] for a high-level view of heap exploits). This type of of **weird programming** has become a staple of modern exploitation (e.g., [14]), and thus must also become a part of defensive security analysis.

1.2 Computational power and pwnage

Hovav Shacham et al. used *computation theory* terms to describe the execution model and computational power of these exploit programming techniques – and showed that they could indeed support Turing-complete environments [40, 38]. Although attackers rarely need such generality to control targets, arbitrarily complex programs such as a Linux kernel rootkit could indeed be compiled and run entirely within the ROP execution model [23].

[1] Some exploitation scenarios also include manipulation of the target's physical environment, such as application of heat, light, or radiation, as well as use of physical side-channels. For the purposes of this paper, however, we only consider scenarios where attacker controls input data of a target's code unit or subsystem.

It would be a mistake to claim that the Turing-completeness of a malicious execution model somehow represents greater practical danger than, say, a finite automaton (so long as the latter, e.g., yields root shell) as it is often the case that the malicious execution is merely used as a stepping block within an attack. Turing completeness is an "estimation from above" of the attacker's power over the execution and a statement that there is no algorithmic task possible on the target environment itself that the attacker could not successfully emulate. "Root shells" in classic Unix exploitation scenarios[2] represent a similar symbolic statement: even though the attackers' goals could possibly be accomplished without spawning a root shell, it serves as evidence of full attacker control. To quote security researcher Felix 'FX' Lindner, "You can't argue with root shell."

1.3 Our contributions

In this paper we introduce the notion of metadata-driven computation and demonstrate the indispensable role metadata have played in exploit-techniques to this day (Section 2). We will then show that metadata can be just as powerful as code or stack ROP chains by presenting the Cobbler toolkit which crafts **ELF** (executable and linking format) metadata to take advantage of the Turing-complete execution environment present in the runtime loader (RTLD). Cobbler produces programs (encoded as ELF metadata) that are interpreted by the `ld.so` runtime loader (RTLD) as the executable that contains these metadata is setup for execution (Section 4). This computation happens with full knowledge of the executable's dynamic symbols and thus is not mitigated by address space layout randomization (ASLR). This opens obvious opportunities for both hiding code-free Trojan logic and obfuscating the program flow.

Although the overall ELF file and each metadata entry in particular are *well-formed*, it is doubtful that the binary toolchain developers expect the RTLD to perform arbitrary memory transformations and other kinds of general computation.

In general we would like to put out a call to arms for researchers to better understand these weird machines and the role metadata, data, and code have in driving such machines.

2 A look at metadata

Metadata (and generally speaking, parameters, options, and other configuration data) fuel software composability, adaptability, and diversity. Most modern general-purpose software is designed to work on a variety of architectures and operating systems, following the philosophy that code should be reused and shared whenever possible.

Metadata also allows processes constructed from the same code units to be laid out differently in memory, forcing attackers to deal with a diversity of targets. Forrest has shown that this metadata-supported diversity may allow systems to be more robust against threats and vulnerabilities [18], and has proven its usefulness in many cases (e.g.,[16, 19]). However we posit that at the same time we must be cautious – rich-enough metadata may expand the attack surface of a system in unexpected ways.

Hacker research and metadata. Given hacker intuitions about computation as the key object of security, it's not surprising that hacker research leads in understanding the role of ABI metadata in modern computational environments. For instance, to date the guides published by Grugq ([44, 21]) and Mayhem ([31]) remain the most detailed guides to runtime ELF ABI, besides Levine's solitary [27] book on the subject. Many Phrack articles [25, 9, 39, 10] illustrate both ABI engineering principles and abuse of their artifacts. Hacker publications, such as the *Corkami* collection [3], provide comprehensive tests of binary formats' edge cases. Hacker research into unexpected but reliable ways of binary composition by leveraging ABI structures has given rise to a number of design patterns [7]. Not surprisingly, the most advanced framework for manipulating ABI metadata, ERESI [2], also originates from hacker research [30, 41].

2.1 Exploits

Although metadata are rarely the main focus of an attack, metadata have long been leveraged as a means of carrying out exploits. Even the original stack-smashing attacks can be said to target control flow metadata in the stack frames [34]; more advanced software exploitation techniques such as ROP and heap-smashing all aim to overwrite metadata such as a function pointer stored somewhere in writable memory waiting to be invoked. Viruses and other types of malware are a classic example of attacks achieved via manipulating metadata: parasitic code injected into an existing executable will never see the light of day unless the executable or library's metadata are changed to inform the loader of its existence and to get some other component in the runtime to eventually invoke the parasitic code.

Data-driven Turing machines. We are not the first to explore data-driven computation. Numerous instances

[2] Prior to parceling out of the all-powerful root's privileges, such as with capabilities, SELinux, Solaris zones, LIDS, etc.

of data-driven Turing machines have surfaced during the past decade and are presented as examples of how computationally-rich the environments processing these metadata are. For example, Eli Fox-Epstein has shown that HTML in combination with CSS3, normally thought of as markup languages, can drive Turing-complete computation [17]. Others have shown how the instructions written on the cards in the game *Magic: The Gathering* can drive a Turing-complete machine [12]. Todd L. Veldhuizen at Indiana University showed that C++ Templates can drive a Turing machine [45]. These are only a few examples of weird-machines driven by instructions that are not normally thought of as true executable code.

2.2 Defenses

Metadata are often underappreciated by security researchers and are often seen as a means to an end – data that can be shaped in fuzzing to search for an application's vulnerabilities or data that can be leveraged to carry out an exploit. And yet we find that software defense practices do not always treat metadata with the respect they deserve. For example, the antivirus industry has long based their virus detection mechanisms on code fingerprints. Although the antivirus industry has made strides towards heuristic runtime virus detection, we still see much effort spent on code signatures. Perhaps the time will soon come to where they consider metadata fingerprints.

Software integrity. A classic technique used in software defense is to check the integrity of a piece of software as it appears on disk before it is loaded and executed. This type of defense and been developed mostly in response the world of viruses, rootkits, and malware but also can be used to determine *who* generated a particular executable or library.

Executable signing is one example of a software integrity mechanism. Various implementations of executable-signing schemes demonstrate that security practitioners think of computation as something that can only be generated from code and thus metadata can be trusted to a small extent. The small amount of implicitly-trusted (unsigned) metadata allowing for flexibility in the signing scheme can end up being the implementation's Achilles' heel.

At REcon 2012, Igor Glücksmann demonstrated how data can be injected into a signed PE file without invalidating the signature [20]. His technique took advantage of the fact that the signature and the signature's metadata embedded in the PE were not themselves signed, and that PE supported variable length signatures where injected data could hide. This support for flexibility and implied trust of some metadata ended up being this signature scheme's weakness.

Various ELF-signing mechanisms have been implemented, each differing on how and what components of the ELF are signed. We discuss two such implementations: `elfsign` and `signelf`.

`elfsign` [42], which used to be available in the Debian package repositories, was designed to sign all ELF metadata except data contained in the signature's section (the section named ".sig"), the null-terminated string ".sig" in the string table, and the the last section header (we assume to be the signature's section header). Although `elfsign` has been retired due to its use of the weak MD5 signature-scheme before any attacks were developed[3] this signing scheme appears to be very similar to the PE-signing scheme, and thus is likely to be vulnerable to attacks similar to those presented by Glücksmann.

`signelf` [13] is another example of an ELF-signing implementation, although it is unclear whether `signelf` is used in practice. `signelf` suffers from an alarmingly obvious weakness – it only signs a specific set of ELF sections that it looks up by name (whereas the dynamic loader does not locate sections by name), and it does not sign the metadata that describe where to find these sections. Therefore section metadata can be changed, and unsigned sections can be modified. This is yet another example of how metadata is underappreciated in practice.

3 Runtime linking and loading

In order to understand the ELF metadata-driven weird machine present in Linux's RTLD we must first understand the steps that are taken to load an executable into memory and then kick off execution. The loading process, starting at the initial call to `exec()`, until the executable starts running is illustrated in Figure 1. New processes are created using the `exec()` system call which is provided with the path of the file to be executed. Calls to `exec()` cause the kernel to read small subset of the executable's metadata in order to map the executable to memory (`hello`, in the case of Figure 1) and the executable's interpreter (typically the RTLD, `ld.so`) into the process' address space. Next a context switch into userland is made and the interpreter is kicked off (`RTLD_START()` is called in the case of `ld.so`). It is the interpreter's job to load any libraries such as `libc.so` and patch memory as specified by the executable's metadata before finally passing off control to the executable's entry-point (typically `_start()`).

[3] `blog.andrew.net.au/2010/01/23`

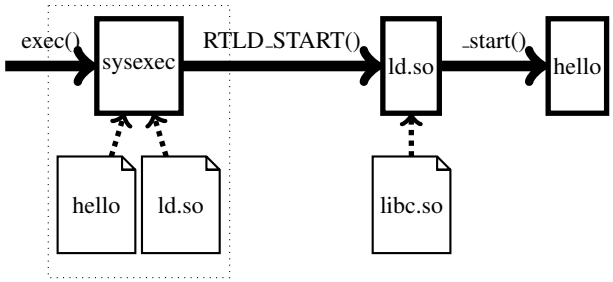

Figure 1: Overview of runtime loading process for an executable named hello.

3.1 Symbol lookup and link_map structures

As the RTLD loads the executable and each library the executable is dependent on, the RTLD creates and maintains one `link_map` data structure for each **ELF object** (library and executable) that is loaded. `link_map` structures are only created at runtime and contain information including:

- The name of the ELF file for which the `link_map` structure corresponds.
- The base address at which the ELF object was loaded.
- The virtual address of all the ELF object's dynamic table entries.
- Pointers to other loaded `link_map` structures so that they all form a doubly linked list.

All the `link_map` structures form a doubly linked list so that as long as we can locate a single `link_map` structure, we can find information on any other loaded ELF object. The ordering of these `link_map` structures is important with regard to symbol resolution – the symbol resolver traverses these `link_map` structures in linear order as needed to locate the information needed to resolve the symbol.

3.2 ELF metadata

The metadata contained in ELF files are the conduit by which the compiler, linker, and loader communicate. The purpose of ELF metadata is to keep track of properties of the machine code it encapsulates. Such properties include libraries that its code depends on and locations of addresses or data in the file that need to be patched in order for the code to cleanly execute once loaded.

The ELF header. All ELF files contain a structure known as the **ELF header** (`Elf64_Ehdr`[4]) at the beginning of the file. The ELF header contains information such the ELF file's type (executable, shared-object, etc.), architecture it was compiled for, and where in the file other metadata can be located. Any particular piece of metadata needed by the linker or loader can be found by crawling the data structures referenced by the ELF header.

[4]We are assuming a 64-bit architecture, ELF metadata for 32-bit architectures begin with `Elf32_`

3.2.1 ELF sections and metadata tables

ELF files contain numerous tables of metadata, each table contained in a single ELF **section**. The ELF header contains information on how to locate ELF section header table. ELF section headers allow us to locate each table of metadata. Each metadata table generally holds metadata of a single type: such as symbol metadata (`Elf64_Sym`) or relocation metadata (`Elf64_Sym`). The vast majority of ELF metadata is stored in one of its many tables.

The dynamic table. All ELF executables and shared-object libraries contain a table of `Elf64_Dyn` structures known as the dynamic table. Each `Elf64_Dyn` structure has a tag and a data field that contains either a pointer or a value. The structure's tag marks what the rest of the structure contains and how it should be interpreted. The dynamic table exists to summarize information required at runtime such as the names of libraries needed as well as the locations and sizes of metadata that the RTLD needs to process. The dynamic table acts as a convenient one-stop shop for the dynamic linker and RTLD, most of the data it contains can be found by crawling through the structures pointed to by the ELF header.

Symbol metadata. Symbol metadata provide meaning and context to data and functions that need to be located at runtime. These metadata typically provide information about objects that are imported or exported such as binding information so that, for example, an executable can call a function defined in an external library.

ELF files contain symbol metadata in the form of `Elf64_Sym` structures. Figure 2 shows the definition of `Elf64_Sym` structures. In summary, symbol structures contain a pointer to the symbol's name, information on the type of symbol, and a value which most often is a pointer to the object (data or function) itself. Symbols of type `STT_IFUNC` are not as simple to interpret as other types of symbols. The purpose of `STT_IFUNC` symbols is to allow the decision of what version of a function to be used to be deferred until load/runtime. A `STT_IFUNC` symbol's value points to code that returns the address of the function it ultimately decides to use.

```
typedef struct {
  Elf64_Word    st_name;  //Index of name
  unsigned char st_info;  //Type  info
  unsigned char st_other; //Not used
  Elf64_Half    st_shndx; //Section #
  Elf64_Addr    st_value; //Sym value
  Elf64_Xword   st_size;  //Size of object
} Elf64_Sym;
```

Figure 2: Contents of ELF symbol metadata, **`Elf64_Sym`**

```
typedef struct {
  Elf64_Addr   r_offset; //Addr to patch
  Elf64_Xword  r_info;   //Type&sym index
  Elf64_Sxword r_addend; //Addend
} Elf64_Rela;
```

Figure 3: Contents of ELF relocation metadata, **`Elf64_Rela`**

Relocation metadata. Relocation metadata provide the linker and loader with information on which virtual addresses should be patched, and how. For example, executables maintain a table of pointers to imported library functions at runtime; some relocation entries are used to patch the table with the locations of these imported functions. Some, but not all, types of relocation table entries make references to symbol metadata to provide extra context on how the patch value should be calculated. Relocation metadata comes in the form of `Elf64_Rela` structures as shown in Figure 3.[5]

ELF executables generally contain two relocation tables, `.rela.dyn` and `.rela.plt`. `.rela.plt` entries are typically processed lazily during dynamic linking. `.rela.dyn` relocation entries are processed during load time after the RTLD has mapped all of the required libraries to memory but before the RTLD passes control to the executable. Any references to symbols made by the relocation-table entries in `.rela.dyn` are encoded as indices into `.dynsym` symbol table in the `Elf64_Rela`'s `r_info` field. The ELF's dynamic table contains the address of both `.rela.dyn` and `.dynsym` sections so that this information can be quickly looked up.

The System V amd64 ABI [28] defines 37 different types of relocation entries, the `gcc` toolchain we worked with only uses 13 types (one of which is not defined in the ABI), whereas our proof-of-concept Cobbler compiler, makes use of only 3 different types of relocation entries: `R_X86_64_COPY`, `R_X86_64_64`, and `R_X86_64_RELATIVE`, which we will abbreviate as `COPY`, `SYM`, and `RELATIVE`.

Table 1 summarizes these three types of relocation entries. `Elf64_Rela`s of type `COPY` instructs the RTLD and dynamic linker to perform what essentially is a call to `memcpy()` where the associated symbol's value points to the bytes to be copied and size contains the number of bytes to copy. `Elf64_Rela`s of type `SYM` request that the value in the addend is summed with the symbol's value and the ELF object's base address (typically 0 for an executable), which is written to the specified offset plus base address. `Elf64_Rela`s of type `RELATIVE` do not make use of any symbols, they simply add the value of the object's base address to their addend and store that in the relocation entry's offset plus base address.

`COPY`	`memcpy(r.offset,s.value,s.size)`
`SYM`	`*(base+r.offset)=s.value+r.addend+base`
`RELATIVE`	`*(base+r.offset)=r.addend+base`

Table 1: Relocation entries Cobbler makes use of and their meaning presented in c syntax. For each type, r is the relocation entry structure and s is the corresponding symbol, and `base` is the base address of where the ELF object is loaded.

4 Cobbler implementation

ELF symbol and relocation entries allow for code adaptability and reuse, however, they can be crafted to perform other types of computation. We have built a proof-of-concept toolkit[6], Cobbler, that compiles the non-I/O related instructions in Brainfuck (an esoteric Turing-complete language [1]) down to ELF metadata and injects the metadata into an executable (Section **??** describes where to find the toolkit). The remainder of this section will describe our implementation of the primitive instructions upon which we built our Brainfuck to ELF metadata compiler to demonstrates how ELF metadata can awaken the Turing-complete weird machine hidden in the RTLD.

4.1 Tools

Our proof-of-concept compiler was built and tested on Ubuntu 11.10's gcc toolchain, `eglibc-2.13`, running on an amd64 architecture. We do not have any reasons to believe that the compiler cannot be ported to work with other gcc versions but we have not attempted to port the our compiler and tools.

Figure 4 shows how an executable "enhanced" with computation-driving metadata – what we will refer to as a **Cobbler-enhanced** executable – is constructed. In order to use Cobbler to take advantage of the RTLD's weird machine, one must have write access to some target/host executable. Given a list of Cobbler-supported instructions and an executable in which to enhance, the Cobbler

[5] There are versions of 64-bit ELF relocation structures that do not contain the addend field (`Elf64_Rel` structures) however we will not consider them in this paper.

[6] available at `http://github.com/bx/elf-bf-tools`

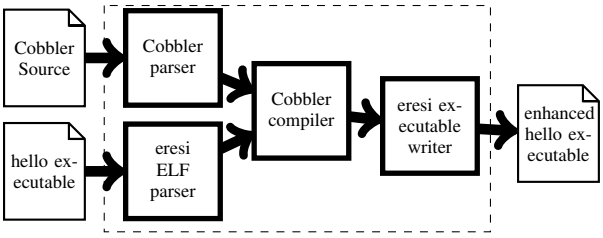

Figure 4: How a Cobbler-enhanced executable is constructed.

compiler first parses the source code that it is compiling, then by using the ERESI toolkit [2] to parse the executable, it crafts metadata to inject and hands the new metadata off to ERESI to create an Cobbler-enhanced copy of the executable. Cobbler carefully constructs its enhancements to allow for clean execution of the executable after the Cobbler-enhanced metadata are processed.

It is important to note that currently only non position-independent (PIC) executables – those whose base location are not randomized – can be Cobbler-enhanced. This is reasonable given that the majority of ELF executables are not compiled to be position-independent by default. We have not attempted to work with PIC executables but have no reason to believe that they are fully resistant to Cobbler. Cobbler does not make any assumptions about where libraries are mapped and thus is unaffected by libraries mapped at random locations by ASLR. It is in fact possible to locate all libraries at load-time using ELF metadata. This technique is described in Section 4.2.4.

4.2 Cobbler primitives

We can think of the primitives implemented in Cobbler as an assembly-like language composed of three basic instructions:

1. Addition (`add`),
2. Move/copy (`mov`),
3. Jump if not zero (`jnz`).

In this language, the bytecode is composed of relocation metadata. Symbol metadata plays the role of registers. These "registers" just also happen to be memory-mapped and also contain inline metadata.

Cobbler primitives use four different addressing modes for its operands, although no single instruction operand supports all addressing methods. The supported addressing modes and syntax we use are as follows:

- **Immediate**: value is specified directly in instruction (e.g. $0x01)
- **Direct**: instruction contains value's address (e.g. *0xdeadbeef)
- **Register**: value is contained in register (e.g. %reg)
- **Register indirect**: register contains value's address (e.g. [%reg])

All destinations are directly addressed across all instructions. The following sections demonstrate how each instruction is implemented.

4.2.1 Move

The `mov` instruction is expressed as

```
mov <dest>, <value>
```

where `<dest>` is specified in direct mode and `<value>` can be specified either as an immediate or register indirect. The `mov` instruction copies the value to the address specified by `<dest>`, the destination. We have implemented `mov` to always copy 8 bytes to its destination, but the number of bytes that are copied can be adjusted.

Consider the following `mov` instruction that uses immediate addressing:

```
mov *0xbeef0000,  $0x04
```

The following relocation entry implements this immediate `mov` instruction:

```
{type=RELATIVE, offset=0xbeef0000,
 symbol=0, addend=0x04}
```

Relocation entries of type `RELATIVE` naturally implement Cobbler `mov` instructions with immediate values. This is because they instruct the linker to copy the value of their addend to the address specified at their offset. (Note that the RTLD ignores ignores symbols when processing `RELATIVE` relocation entries.)

Consider the following `mov` instruction that uses register indirect addressing:

```
mov *0xbeef0000,  [%foo]
```

A relocation entry and symbol table entry are both needed to support this instruction. In this example the relocation entry should be setup as follows:

```
{type=COPY, offset=0xbeef0000,
 symbol=foo, addend=0}
```

This relocation entry makes reference to the symbol `foo`, which is consulted when the relocation entry is processed. The symbol's type is set as an FUNC (and not an IFUNC) so that it is treated as a regular symbol. The symbol's size is 8, this ensures that exactly 8 bytes are copied.

```
{name=foo, value=0xb0000000,
type=FUNC, shndx=1, size=8}
```

This symbol and relocation entry pair instruct the loader to copy 8 bytes starting at `0xb0000000`, the symbol's value, to `0xbeef000`, the relocation entry's specified offset, like `memcpy()`

4.2.2 Addition

The add instruction is written as

```
add <dest>, <addend1>, <addend 2>
```

where <dest> is specified in direct mode, <addend 1> is a register, and <addend 2> is specified as an immediate. add adds the 8 byte <addend 2> to the 8 byte value in the registered specified by <addend 1> and stores that 8 byte result at the address specified by <dest>.

Consider the following add instruction:

```
add *0xbeef0000, %foo, $0x02
```

The following relocation entry implements this instruction:

```
{type=SYM, offset=0xbeef0000,
 symbol=foo, addend=2}
```

A SYM typed relocation entry instructs the loader to copy in the specified symbol's value (in this case foo). It is important that the corresponding symbol's type be a standard, non-IFUNC, type so that the symbol's value is simply treated as a value. The symbol may look as follows:

```
{name=foo, value=1,
type=FUNC, shndx=1, size=8}
```

Such a relocation entry and symbol table entry pair instruct the loader to add the relocation entry's addend (2) to the symbol's value (1) and store the result (3) at the relocation entry's offset (0xbeef0000).

4.2.3 Jump if not zero

The jnz instruction is written as

```
jnz <dest>, <value>
```

where the jump destination, <dest>, is specified as an immediate and <value> is specified in direct mode. Unlike the mov and add instructions, jnz cannot be implemented cleanly with a single relocation entry, this is because the loader was not designed to arbitrarily jump over relocation entries. In this section we will only highlight the major issues encountered when implementing jump instructions, other complications will be discussed in Section 4.3.

In order to understand how such functionality can be implemented using ELF metadata we must understand the context in which relocation entries are processed. The pseudocode in Figure 5 represents a high level/simplified algorithm of how relocation entries are processed by the RTLD.

While processing relocation entries, the RTLD walks through the list of link_map structures (via lm = lm->prev) starting at the last link_map structure on the chain and processes each ELF object's link_map relocation entries in the chain. For each link_map structure, the RTLD looks up the location (lm->dyn[DT_RELA]) and size (lm->dyn[DT_RELASZ]) of the object's relocation table then processes each relocation table entry.

```
while (lm != NULL) {
  r = lm->dyn[DT_RELA];
  end = r + lm->dyn[DT_RELASZ];
  for (r ; r < end; r++) {
    relocate(lm, r, &dyn[DT_SYM]);
  }
  lm = lm->prev;
}
```

Figure 5: Simplified representation of how RTLD processes relocation tables.

We first demonstrate how to implement an unconditional jump instruction before we demonstrate the more difficult task of conditionally branches.

Jump. There are several tasks relocation entries need to perform in order to be able to perform an unconditional branch:

1. Set the value of lm->prev so that the same relocation table is processed on the next while loop iteration.
   - The original lm->prev value needs to be restored later to allow the executable to eventually run
2. Set the value of lm->dyn[DT_RELA] to point to the the jump's destination (relocation entry)
3. Update the size of lm->dyn[DT_RELASZ] to reflect the "new" relocation table size
4. Clobber the value of end so that the RTLD does not process the next relocation entry

Step 1 requires knowledge of where the executable's link_map structure is mapped. It turns out that the RTLD stores a pointer to a dynamically linked executable's link_map in a table the dynamic linker is dependent on – the **global offset table** (GOT). The virtual address of this table is known at compile time (the DT_PLTGOT field in the executable's dynamic table). Using the address of a pointer to the executable's link_map structure we can calculate the address of that structure's prev value. A simple move instruction will implement this for us:

```
mov *<addr of prev>, $<addr of link_map>
```

Writing instructions to restore the executable link_map's prev value is trivial because this

link_map is always the head of the list and thus merely needs to be set to 0 allow the RTLD to continue.

Steps 2–3 requires knowledge of the virtual address of the executable's dynamic table. The executable's entire dynamic table is mapped into memory at the address specified by metadata present in the ELF. Thus we can calculate the virtual address of any item in the executable's dynamic table at compile time. The same holds true for the location of the executable's relocation and symbol tables. Therefore it is trivial to construct a relocation entry that essentially implements:

```
mov *(<address of DT_RELA>), <address
  of next relocation entry to process>}
```

The same applies to step 3.

Step 4 requires the knowledge of the address of end so that we can set it to a value that will cause the next iteration of the loop to exit. It turns out that end is stored on the stack. Because the location of stacks are randomized, calculating the address of end is not so simple. However,the loader stores the address of some stack-allocated data in a statically-allocated variable (_dl_auxv). As described in Section 4.2.4, we can lookup the base-address of the loader to calculate the location of this static variable. The location of end is at a fixed distance from the data _dl_auxv points to, thus using mov instructions we can calculate the address of end and store this value in a symbol. Assuming the address of end is stored in a symbol called sym-end, the following instructions will set end to 0 forcing the loop to exit otherwise prematurely:

```
mov *<addr of next Rela's offset>, %sym-end
mov *<(value overwritten)>, $0
```

Note that the first instruction modifies the second instruction's destination at runtime, copying the address of end found in the sym-end register over to the destination of the second instruction so that when the second instruction executes, its immediate value, 0, will be written to end, forcing the RTLD to stop processing relocation entries.

Given that the dynamic table and link_map are manipulated by steps 1–3 before end is overwritten, the stage is set for a branch before the RTLD quits processing the current round of relocation entries. Once the RTLD quits processing the table, it will attempt to process relocation entries of the previous link_map which now points to the same link_map (due to step 1). The RTLD is none-the-wiser so it looks at the link_map's dynamic table for the addresses of the relocation table to process which now points to the "instructions" of our choosing (from steps 2–3).

Conditional branch. Now that we know how to implement an unconditional branch, we just need to implement a few more instructions to allow for *conditional* branching. The trick to implementing conditional branching lies in how the RTLD handles symbols of type IFUNC. It turns out that if the RTLD is interpreting a symbol of type IFUNC (an indirect function) there are two ways that that it may handle the IFUNC: (1) If the shndx field of the symbol is not 0, then the RTLD treats the symbol as an indirect function, calling the function it points to and using the value returned by the indirect function. (2) If the shndx field is zero, then that symbol's value is used directly. Therefore conditional branches requires a special symbol (we call sym-zero) as well as special Cobbler instructions to initialize the environment for branching. sym-zero is a symbol type IFUNC whose value points to executable code that simply returns 0. Such a gadget can be found in ld.so which is mapped at a higher address than the executable's metadata. We can write the gadget's offset from the base of ld.so into sym-zero's value at compile-time and include Cobbler instructions that can locate ld.so (using the technique discussed in Section 4.2.4) and set sym-zero's value with the sum of its former value (the offset of the gadget that returns 0) and the base address of ld.so.

We then use the following instructions to perform the conditional the branch:

```
mov *<addr of sym-zero shndx>, $<test val>
add *<addr of end>, %sym-zero, $0
```

With this setup, if the value being tested is 0, then the value of the function pointer in sym-zero is written to end, a value which we picked to be larger than the address of the executable's relocation entries, so relocation entry processing continues. We can insert instructions immediately following the jnz bytecode that reset the value of end in case the branch is not taken. If the value we are testing is not 0, then the symbol will be treated as an indirect function rigged to return 0, thus 0 will be written to end forcing a branch.

4.2.4 Locating libraries

We must be able to locate the base address of ld.so to implement conditional branching as described in Section 4.2.3. It can also be useful to locate other libraries too. In this section we describe a technique that uses Cobbler to locate the base address of *any* library loaded by the RTLD. As described in Section 4.2.3, it is relatively trivial to locate the address of the executable's link_map structure. Given that all link_maps are in a doubly linked list, and each link_map contains the base address in which it's ELF object was loaded, we can use the executable's link_map to locate *any* library that the RTLD loaded. Given the executable's link_map, we:

1. Dereference the pointer we have to the to get the base address of the `link_map`
2. Calculate the address of its `next` field (by adding 0x18 to the address of the `link_map`)
3. Repeat steps 1 and 2 until we arrive at the `link_map` of the library we are trying to locate
4. Deference the pointer we have to the to get the base address of the next `link_map`
5. Copy the value at the beginning of the `link_map` structure to a register to be used later (by dereferencing the pointer). This value is the library's base address.

Assuming the library we are trying to locate is the second item on the `link_map` chain and that there is a `sym-lm` register that initially contains the virtual address of the executable's `link_map` structure (known at compile time), the address of the library will be found in the `sym-lm` register after following four instructions are executed:

```
1. mov *<address of sym-lm's value>,
   [%sym-lm]
2. add *<address of sym-lm's value>,
   %sym-lm, $0x18
3. mov *<address of sym-lm's value>,
   [%sym-lm]
4. mov *<address of sym-lm's value>,
   [%sym-lm]
```

4.3 Implementation challenges

This section lists the various challenges we encountered in our implementation of Cobbler primitives while still allowing for clean execution after the RTLD passes control off to the executable's entrypoint.

Preserve existing metadata. In order to allow for the executable to cleanly execute after the RTLD interprets the Cobbler instructions, we must be careful to preserve the existing metadata and to not allow the original set of relocation entries to be processed more than once. The Cobbler-enhanced executable is configured so that our Cobbler instructions are interpreted before the original relocation metadata. The last several relocation entries we inject instruct the RTLD to process the original relocation entries next before processing a final set of relocation entries that restore the `link_map` structure to its original state.

Do not let sanity checks get in the way of branching. The RTLD has some sanity checks as it processes its metadata. Two sanity checks in particular get in the way of branching. One of these tests a boolean stored in the `link_map` structure to see if the relocation entries have been processed yet. It is possible to play tricks with pointers maintained by the `link_map` to get the RTLD to reset the boolean before it attempts to reprocess the `link_map`'s relocation table. A second sanity check causes the RTLD to set the relocation entries as read-only. The workaround is fairly simple and involves setting a particular field in the `link_map` to zero before branching.

4.4 A Cobbler-built backdoor

To demonstrate how Cobbler can be used as an attack vector, we have used Cobbler to insert a proof-of-concept metadata backdoor into the implementation of ping found in Ubuntu's inetutils v1.8. We will briefly describe how a root shell backdoor was inserted into this version of ping. It is interesting to note that we did not need to harness the full computational power to these metadata to insert the backdoor which shows that even relatively weak weird machines can be interesting.

There are two features of this `ping` implementation we make use of to construct a backdoor:

1. `ping` runs `setuid` as root, but drops its root privileges early on
2. The optional `--type` command-line argument takes a single argument to customize the type of packets sent. If provided, `ping` tests the argument in the following manner:

```
if(strcasecmp (<string>, "echo") == 0) {
```

To get `ping` to execute arbitrary programs as root, we must insert metadata capable of doing the following at runtime:

1. Override the call to `setuid()` with something that doesn't produce any noticeable side effects, such as `getuid()`
2. Override the call to `strcasecmp()` with `execl()` so that the `exec` system cal is made instead.

These two actions, in combination, will cause `ping` not to drop root privileges and to treat the argument to `--type` (if provided) as a path to an executable in the call to `execl()` that replaced `strcasecmp()`. If the `--type` option is not provided to `ping`, `ping` still cleanly performs its pinging duties.

Both `setuid()` and `strcasecmp()` are functions that imported from `libc`. The compiler building the `ping` executable does not know where these functions will live at runtime and thus creates entries in the executable's GOT for each function which get lazily filled in by the dynamic linker once their addresses are known. If we fill in the GOT entry for a function before the dynamic linker, the address we provide will be assumed to

be the function's location in memory. The location of the GOT table in memory is known at compile time as are the offsets of `getuid()` and `execl()` from the base address of `libc`. Armed with this information, we can use the library location trick described in Section 4.2.4 to craft metadata that lookup the base address of `libc`, then calculate the absolute addresses of `getuid()` and `execl()`, and finally patch `ping`'s GOT in memory before `ping` is finally executed. It turns out that this backdoor behavior can be implemented using nine relocation entries and one symbol table entry and without making *any* changes to the executable segments of `ping`. You can build a backdoored version of ping using our scripts provided in `elf-bf-tools` in order to see these relocation entries for yourself.

5 Crafting metadata in other types of executables

Although we have mostly focused on ELF metadata, other types of executable metadata have interesting computational influences over their RTLDs.

5.1 PE

In 2006, Skape published a paper called LOCREATE studying novel ways to craft metadata in Windows PE executables [43] LOCREATE's PE metadata crafting technique work as a code unpacking mechanism for malicious software and are more difficult to analyze than traditional code-based malware unpackers. Skape shows how well-formed PE metadata can be crafted to tell the RTLD to overwrite code already loaded in memory. Thus the code that eventually executes does not resemble the code as it appears on disk.

5.2 Mach-O

Mach-O is the executable file format used in Mac OS X. Mach-O stores its relocation metadata at the end of the file with its other dynamic linker metadata. Given that its relocation metadata is located at the end of the file, injecting metadata is relatively straightforward. Mach-O relocation metadata is not stored in fixed-sized structures like ELF and PE but instead is "compressed" into special variable-length bytecode/instructions. For example, the following stream of relocation instructions are used to instruct the linker to patch a particular segment's offset with the address of some library's `setuid()`:

```
1. SET_DYLIB_ORDINAL_IMM <# of library to
   search>
2. SET_SEGMENT_AND_OFFSET_ULEB <segment #>
   <offset>
3. SET_SYMBOL_TRAILING_FLAGS_IMM <flags>
   "setuid"
4. DO_BIND
5. DONE
```

Note that the name of the symbol and library to search is embedded within the relocation instructions.

Imagine an executable that calls `setuid()` to release its administrative privileges. If we change the string `setuid` to `getuid`, a single byte edit, the dynamic linker will link and call `getuid()` instead of `setuid()`, preventing the process from releasing administrative privileges. This small metadata edit has a significant effect on the processes' execution. Mach-O binding instructions available to the RTLD and dynamic linker build a fairly rich language and we suspect that there may be more interesting weird machines hiding there.

6 Future Work

Our analysis of execution models driven by metadata is by no means exhaustive; it is merely meant to attract attention to their computational power and their crucial part in the chain of tools currently implied to "just work" for a computing environment to be trustworthy. Our computation model-based approach to the phenomenon of a "malicious" computation that could be performed with well-formed metadata alone raises important formal questions:

- What should be the trust model and role of metadata in a trusted tool chain?
- Whereas static analysis of code is fraught with halting problem-related challenges, could static analysis of metadata be more satisfactory?
- What properties of currently existing metadata can be formally validated?
- What formal properties should metadata be designed to satisfy so that the computation it drives is trustworthy and validatable?

7 Conclusion

Modern ABI metadata provide powerful composability and diversity benefits for both software engineering in general and security in particular. However, they also developed into a powerful execution environment in which well-formed crafted metadata can drive "weird machines" that transform, with Turing-complete power, the code and data of a process. In designing a trusted toolchain to guarantee trustworthy execution of a process, binary code should no longer be considered the sole seat of computation. We expect that linkers and loaders will become the next nexus of trust in engineering binary tool chains; urgent reflection on the computations they perform is needed if we are to continue to trust them.

8 Acknowledgments

This research was supported in part by the Department of Energy under Award No. DE-OE0000097 and by Intel Lab's University Research Office. The views and opinions in this paper are those of the authors and do not necessarily reflect those of any of the sponsors.

References

[1] Brainfuck. http://esolangs.org/wiki/brainfuck.

[2] ERESI Project. http://www.eresi-project.org.

[3] ALBERTINI, A. Corkami reverse engineering & visual documentations. http://code.google.com/p/corkami/.

[4] ANONYMOUS AUTHOR. Once upon a free(). *Phrack* 57:9. http://phrack.org/issues.html?issue=57&id=9.

[5] ARGYROUDIS, P., AND KARAMITAS, C. Heap Exploitation. Abstraction by Example. OWASP AppSecResearch, 2012. http://census-labs.com/media/heap-owasp-appsec-2012.pdf.

[6] BRATUS, S. Hackers and Computer Science: What Hacker Research Taught Me. 27th Chaos Communications Congress, December 2010. http://events.ccc.de/congress/2010/Fahrplan/events/3983.en.html.

[7] BRATUS, S., BANGERT, J., GABROVSKY, A., SHUBINA, A., BILAR, D., AND LOCASTO, M. E. Composition Patterns of Hacking. The First International Workshop on Cyberpatterns Unifying Design Patterns with Security, Attack and Forensic Patterns, July 2012.

[8] BRATUS, S., LOCASTO, M. E., PATTERSON, M. L., SASSAMAN, L., AND SHUBINA, A. Exploit Programming: from Buffer Overflows to "Weird Machines" and Theory of Computation. *;login:* (December 2011).

[9] CESARE, S. Runtime Kernel kmem Patching. http://althing.cs.dartmouth.edu/local/vsc07.html.

[10] CESARE, S. Shared Library Call Redirection via ELF PLT Infection, Dec 2000.

[11] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium* (2005), pp. 177–192.

[12] CHURCHILL, A. Magic Turing Machine v5: Rotlung Reanimator / Chancellor of the Spires. http://www.toothycat.net/~hologram/Turing/HowItWorks.html.

[13] CODEFOX. SignElf. http://sourceforge.net/projects/signelf/.

[14] DULLIEN, T. Exploitation and state machines: Programming the "weird machine", revisited. In *Infiltrate Conference* (Apr 2011).

[15] DULLIEN, T., KORNAU, T., AND WEINMANN, R.-P. A Framework for Automated Architecture-Independent Gadget Search. In *USENIX WOOT* (August 2010).

[16] EAGLE, C. Ripples in the Gene Pool - Creating Genetic: Mutations to Survive the Vulerability Window. Defcon 14, August 2006.

[17] ELITHEELI. "stupid machines". https://github.com/elitheeli/stupid-machines.

[18] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)* (Washington, DC, USA, 1997), HOTOS '97, IEEE Computer Society, pp. 67–.

[19] GEER, D. CyberInsecurity: The Cost of Monopoly. Computer and Communications Industry Association (CCIA) report, 2003.

[20] GLÜCKSMANN, I. Injecting custom payload into signed Windows executables Analysis of the CVE-2012-0151 vulnerability. ReCON, June 2012. http://recon.cx/2012/schedule/events/246.en.html.

[21] GRUGQ, AND SCUT. Armouring the ELF: Binary encryption on the UNIX platform. *Phrack* 58:5. http://phrack.org/issues.html?issue=58&id=5.

[22] HUKU, AND ARGP. The Art of Exploitation: Exploiting VLC, a jemalloc Case Stud y. *Phrack Magazine 68*, 13 (Apr 2012).

[23] HUND, R., HOLZ, T., AND FREILING, F. C. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium* (2009), USENIX Association, pp. 383–398.

[24] JP. Advanced Doug Lea's malloc Exploits. *Phrack* 61:6. http://phrack.org/issues.html?issue=61&id=6.

[25] KLOG. Backdooring Binary Objects. *Phrack* 56:9. http://phrack.org/issues.html?issue=56&id=9.

[26] KORNAU, T. A gentle introduction to return-oriented programming. http://blog.zynamics.com/2010/03/12/, March 2010. Zynamics blog.

[27] LEVINE, J. *Linkers and Loaders*. The Morgan Kaufmann Series in Software Engineering and Programming, 1999.

[28] MATZ, M., HUBICKA, J., JAEGER, A., AND MITCHELL, M. System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.96, June 2005. http://www.uclibc.org/docs/psABI-x86_64.pdf.

[29] MAXX. Vudo malloc Tricks. *Phrack* 57:8. http://phrack.org/issues.html?issue=57&id=8.

[30] MAYHEM. The Cerberus ELF Interface. *Phrack* 61:8. http://phrack.org/issues.html?issue=61&id=8.

[31] MAYHEM. Understanding Linux ELF RTLD internals. http://s.eresi-project.org/inc/articles/elf-rtld.txt, Dec 2002.

[32] NERGAL. The Advanced return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine 58*, 4 (Dec 2001).

[33] OAKLEY, J., AND BRATUS, S. Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code. In *USENIX WOOT* (2011), pp. 91–102.

[34] ONE, A. Smashing the Stack for Fun and Profit. *Phrack* 49:14. http://phrack.org/issues.html?issue=49&id=14.

[35] PATTERSON, M. L., AND BRATUS, S. The Science of Insecurity. 28th Chaos Communications Congress, December 2011. http://langsec.org/.

[36] REDPANTZ. The Art of Exploitation: MS IIS 7.5 Remote Heap Overflow. *Phrack Magazine 68*, 12 (Apr 2012).

[37] RICHARTE, G. Re: Future of Buffer Overflows. Bugtraq, October 2000. http://seclists.org/bugtraq/2000/Nov/32.

[38] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.

[39] SD, AND DEVIK. Linux On-the-fly Kernel Patching without LKM, Dec 2001.

[40] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: return-into-libc without Function Calls. In *ACM Conference on Computer and Communications Security* (2007), pp. 552–561.

[41] SHELL CREW, T. E. Embedded ELF Debugging : the middle head of Cerberus. *Phrack* 63:9. http://phrack.org/issues.html?issue=63&id=9.

[42] SKAPE. ELF binary signing and verification. ttp://www.hick.org/code/skape/papers/elfsign.txt, January 2003.

[43] SKAPE. Locreate: an Anagram for Relocate. *Uninformed 6* (Jan 2007).

[44] THE GRUGQ. Cheating the ELF: Subversive Dynamic Linking to Libraries. althing.cs.dartmouth.edu/local/subversiveld.pdf.

[45] VELDHUIZEN, T. L. C++ Templates are Turing Complete. http://ubietylab.net/ubigraph/content/Papers/pdf/CppTuring.pdf. Indiana University Computer Science.