4-18-2011

Exploiting the Hard-Working DWARF: Trojans with no Native Executable Code

James Oakley
Dartmouth College

Sergey Bratus
Dartmouth College

Dartmouth Computer Science Technical Report TR2011-680

Exploiting the Hard-Working DWARF: Trojans with no Native Executable Code

James Oakley and Sergey Bratus
Computer Science Dept.
Dartmouth College
Hanover, New Hampshire
`james.oakley.11@alum.dartmouth.org`

April 18, 2011

Abstract

All binaries compiled by recent versions of GCC from C++ programs include complex data and dedicated code for exception handling support. The data structures describe the call stack frame layout in the DWARF format bytecode. The dedicated code includes an interpreter of this bytecode and logic to implement the call stack unwinding.

Despite being present in a large class of programs – and therefore potentially providing a huge attack surface – this mechanism is not widely known or studied. Of particular interest to us is that the exception handling mechanism provides the means for fundamentally altering the flow of a program. DWARF is designed specifically for calculating call frame addresses and register values. DWARF expressions are Turing-complete and may calculate register values based on any readable data in the address space of the process. The exception handling data is in effect an embedded program residing within every C++ process. This paper explores what can be accomplished with control of the debugging information without modifying the program's text or data. We also examine the exception handling mechanism and argue that it is rife for vulnerability finding, not least because the error states of a program are often those least well tested. We demonstrate the capabilities of this DWARF virtual machine and its suitableness for building a new type of backdoor as well as other implications it has on security.

Contents

Chapter 1

Introduction and Background

1.1 Introduction

This chapter offers a general overview of the ELF structure, its history of use in exploitation, and an overview of our contributions. Those who wish to jump into the technical details may wish to skip to Chapter 2. Historically, exploitation mostly focused on the "main" computation performed by the code of the target program and the libraries loaded into its process context (for the sake of the argument, let us define this computation as the flow described by the target program's call graph). In ELF[1] terms, it was the contents of `.text` sections of executable and shared object files that received the most attention (such as being scanned for vulnerabilities, trojan logic, or "gadgets" to aid exploits).

However, a typical ELF process context is also constructed and maintained by a number of what we would call *auxiliary computations*, driven by data and/or code from other, non-`.text` ELF sections. These computations handle the special stages of the target process' lifecycle, from creation and initization, loading and relocation, to dynamic loading and linking in of required libraries and library functions (thus actually connecting the function-level call graph), to exception handling (the focus of this paper) and process dismantling. The sections involved include relocations sections `*.rel*`, dynamic symbol tables and other resources pointed to the directory that is the `.dynamic` segment, the pairwise related `.init`, `.fini` and `.ctors`, `.dtors`

[1]Executable and Linkable Format

sections, and so on.[2]

Whereas most of these auxiliary computations seem trivial, e.g., `.init` and `.fini` merely iterate over the function pointer tables in `.ctors` and `.dtors`, they may already present convenient interfaces for exploitation (e.g., [26, 17]). More complex auxiliary computation subsystems such as those responsible for dynamic linking present much richer targets [7, 33, 22], allowing for advanced techniques that co-opt the entire subsystem's functionality, such as the pioneering PaX non-executable memory emulation and address load randomization bypass techniques [21], which, among other contributions, had co-opted the dynamic linker itself for resolution of load-time randomized addresses!

Thus exploiting auxiliary computations — those not driven by the target's `.text` sections — has a history, on which we will comment further in Section 1.5.

We view expoitation uses of the auxiliary computations as facets of a single general phenomenon, that of programming automata responsible for these computations with data contained in the appropriate ELF sections. These sections are interpreted by their respective automata, and, when filled with crafted "program", can leverage their computational power to accomplish a lot more that intended by their original designers and programmers, all the the while not necessarily breaking the intended semantics of the automaton's interpretation of the crafted section contents. Instead, the crafted program is abusing the additonal flexibility that designers put into the automaton.

In this paper we continue this vein by demonstrating the use of the DWARF-based exception handling mechanism — the most flexible such mechanism by design, ubiquitous in modern GCC-build environments — to host a Turning-complete Trojan computation. Our Trojan "program" for the DWARF exception handling "machine" is composed entirely of valid DWARF virtual opcodes and contains no native x86 or other platform code.

One of these auxiliary computation mechanisms is the exception handling mechanism. There are several reasons why attacking exception handling is attractive. If a backdoor triggered only by exception handling is placed it may be difficult to detect. Aside from being unexpected (since this novelty will of course eventually wear off), it runs a very low risk of disrupting normal program flow. If one chooses an exception which is triggered very rarely in

[2]Besides the classic `.text`, `.data`, `.rodata`, `.bss` and others, a modern ELF file may include over 30 interrelated sections with semantically distinct functionality.

normal program operation (and indeed, in a well-engineered program any exception will fit that criterion), then there is very low risk of the backdoor being triggered/revealed unintentionally. The triggering mechanism is also already built in. When coming from the other side (i.e. not a backdoor but a direct attack on running software) exception handling is attractive because it is often not well tested. While good testing always tests the error states of a program as well as the operational states, in actual practice the error states are often tested only cursorily if at all since functionality is viewed as most important and because unlike operational states, error states are not tested upon every trial run of the software. Therefore, the likelihood of finding bugs (some of them constituting vulnerabilities) within these regions of a program is increased.

1.2 Contributions

We present a further step in the direction of utilizing the "auxiliary computations" to accomplish unintended, potentially malicious goals, using the arguably most powerful (in fact, Turing-complete) ubiquitous auxiliary environment to date, the DWARF exception handling mechanism.

In particular, we present a way of programming the DWARF exception-handling virtual machine environment that comes with every modern GCC-compiled exception-aware executable or shared objects files by way of providing it with crafted contents of the `.eh_frame` and `.gcc_except_table`.

We show that these contents, originally meant to flexibly and extensibly accommodate present and future stack unwinding and saved register restoration logic, should be understood as powerful bytecode that allows execution of generic computations that, among other things, can read the main process memory, and in doing so make full use of the target's dynamic symbol information.

Moreover, the bytecode is in fact very efficient at representing such symbolic memory operations, and allows us to pack much functionality into short snippets of bytecode. For example, we can package our own self-contained dynamic linker into less than 200 bytes, which easily fits within the typical `.eh_frame` section length, sparing us the effort of rebasing or relocating any other sections.

We note that this mechanism

1. involves no native executable binary code, and therefore is easily *portable*

between systems using the same or binary compatible versions of the standard exception handling libraries;

2. for the same reason, is unlikely to be checked by any current signature-based HIDS systems;
3. is *ubiquitous*, since it occurs where ever GCC-compiled C/C++ code or other exception-throwing code is supported;
4. can easily adapt to different versions of standard libraries, since the prevalence of a few standard GNU/Linux distributions means that there are only a handful of common builds, and the existing ones can be tested by universally supported code prior to deploying the attack payloads;
5. bypasses ASLR by using the system's own facilities;
6. if combined with an appropriate memory corruption bug, can most likely be made into a non-traditional exploit payload such as a shellcode or better;
7. is unconstrained by the technical limitations of return-oriented-programming (ROP) and similar techniques that depend on a careful chaining of multiple borrowed native code gadgets: once control is given to the crafted DWARF "program" as a result of an exception, any values can be prepared and any computation can be done entirely from the DWARF virtual machine itself.

We have modified Katana, our existing academic ELF-manipulation tool to allow us to demonstrate the techniques we discuss.

1.3 History and Prior Work

This section sketches the history of the native binary code's changing role in exploit programming, from straight shellcoding to "return-oriented" and other code-borrowing techniques. It then describes the alternative approaches that, unlike the latter, do not fragment the semantics of the target's code units but rather make use of computations afforded by these units in their entirety, in effect acting as "programs" for the automata implemented by these units, not far from the implementor's original semantics.

For example, while ROP "gadgets" are selected from the target's loaded code — `.text` of a library, the target process, or OS kernel — without any regard for the containing unit's semantics other techniques like [21] or [28] use original developer-intended granularity components of a loaded process image specifically for the kind of computations they were meant to provide (although not in the contexts they were meant for).

The readers already intimately familiar with these techniques are encouraged to skip to Section 2.

1.4 Exploitation at native binary code granularity

Historically, the prevailing notions of computer system exploitation tended to revolve around its platform's native binary code, at the granularity of single binary or assembly instructions.

The hacker research community has long followed the paradigm of approaching exploitation as a kind of (macro) assembly programming [25, 10, 21, 24, 9] based on co-opting the binary code of the target through its (exploitable) bugs such as memory corruptions [19].

Whereas other kinds of exploitable bugs such as integer overflows, escape character (mis)interpretation, Unicode parsing ambiguities, internal command language injection (shell commands into CGI scripts, SQL injections, etc) are also recognized as valuable, the popular judgment of technical supremacy is clearly given to exploits that deliver full programmatic control of the target.

This degree of control tended to circle back to the ability to execute the binary code of attacker's choice. Intially, this binary code existed as *shellcode*, an executable binary blob snuck into the target process' address space by means of crafted input, and placed in the path of the CPU's control flow in the context of the target process.[3]

Following the introduction of early non-executable memory countermeasures [32, 23] (i.e., prior to the adoption of the MMU-supported non-executable (NX) bit page-level tagging by the CPU/MMU vendors), the focus of ex-

[3]Kernel exploits, such as network link layer driver exploits [2], display the same basic approach, even though they must build on the kernel's internal API or DDK when defined, rather than on the well-specified system call or standard library APIs.

ploitation research shifted to "borrowing" necessary executable code snippets from the target's own address space [29, 37], finally achieving academic recognition as a general, Turing-complete technique through [27, 6, 14] and subsequent developments.

Still, the emphasis of this direction is on programming with *native* binary code at instruction granularity, whether injected or borrowed. In particular, successful exploitation required knowledge of, access to, and building on long chains of exact binary or assembly snippets. In contrast, if DWARF data can be injected it can directly perform computation. Instead of a target program being supplied with data to chain snippets of code painfully found in the original program, only the environment to throw an exception must be found in the target program once data injection is achieved.

1.5 Exploiting the auxiliary computations

However, side-by-side with instruction-granular techniques, another approach has been developing. It recognized that large units of code — spanning multiple functions and/or system calls — already present in the target by the original software engineering design could be used to perform computations of interest to the attacker by merely manipulating the input data structures to these units.[4]

This approach yielded rich results when applied to the omnipresent standard code that performed the auxiliary computations in the target process' lifecycle. As those computations necessary to create, load, link, debug, handle exceptions, and finally dismantle a process got progressively more complicated with the progress of operating systems, compilers, and programming environments — the number of ELF sections typically present in a binary nearly doubled in the last decade — the subsystems that perfomed them got both more powerful and better defined.

As a result, many of these subsystems, for example the relocation subsystem mentioned below and the exception handling subsystem that is the subject of this paper, developed into well-defined automata with the input

[4]Arguably, this description could apply to individual system calls as used by a classical shellcode or to library calls chained by a sequence of crafted stack frames in the style of [21], if one takes into account their decomposition into internal sub-units. However, we attach significance to their authors' idea of what constituted a natural functionality decompositon and made these units "elementary" as opposed to larger "subsystems".

data formats what amounted to their own distinct sets of virtual instructions.

These excellent software engineering developments have been noticed and co-opted by the exploit developers. The target's code functionality of these automata could now be borrowed not just as a matter of opportunity or convenience orthogonal to their original purpose, but rather for their original, intended function. This was in stark contrast to "return-into-lib" exploit "gadgets" and other pieces of borrowed logic that accomplished a task conducive to the exploitation such as a memory overwrite — say, as the *malloc* implemmentation's doubly-linked list patching code that manages the allocated block list is commandeered to perform a memory overwrite in a "double-free" exploitation scenario [4] — and offered superior reliability of exploits and even the power to bypass protections like ASLR (e.g., [21, 10]).

1.6 LOCREATE: a case study in programming the standard relocator.

A great – and pedagogically perfect in its clarity — example of using a load-time auxiliary computation for a task traditionally performed by native binary code is provided by the LOCREATE unpacker [28]. LOCREATE uses the ELF process context's own relocation mechanism as the unpacker, driving it with crafted relocation sections. In effect, it treats the relocation subsystem as a distinct memory-transforming automaton that happens to be present in the target's context, and drives the code transformation with a crafted "program" for this automaton.

This method is in striking contrast to the traditional unpacking of injected code (packed for inconspicuous transport to avoid NIDS and HIDS signature detection), which has been traditionally handled by custom snippets of native code (themselves prone to detection). Advanced target process instrumentation techniques such as Ollybone [30] arose to catch the unpacked code just after the unpacker was done with it, right at the point when the unpacked code started executing.[5] LOCREATE's proof-of-concept unpacking performed by the relocation subsystem following the supplied "relocation

[5]We regard Ollybone's manipulation of the x86 hardware memory translation features as a first example and a harbinger of smarter memory trapping to catch composite events, in particular "page just written to was used to fetch executable code from", expressing a two-step trappable memory event composed from a temporal, sequential relationship of elementary memory events "read, write, execute".

program" requires no injection of native executable code whatsoever to accomplish the task.

Chapter 2

Technical Background

In order to understand both how the exception-handling process may be controlled to engineer an exploit and the functionality of our tool Katana it is necessary to understand how the C++ exception handling process as implemented by GCC and as partially standardized by the Linux Standards Base [1] and the x86_64 ABI [20]. All technical details are discussed with regards to C++, GCC and Linux and with specific attention paid to the x86_64 architecture. The concepts (and most of the details) apply equally well to other processor architectures and to the BSDs, Solaris, and most other Unix/Unix-like systems where GCC is used. In addition, the Clang C++ compiler is known to be (nearly) fully binary compatible with GCC including with largely undocumented GCC language/implementation-specific exception handler tables, as can be seen in the LLVM source [16]. Further, the exception-handling data formats and processes are not C++-specific. All study in this work has been in the context of C++ but the same general method is used for other gcc-compiled languages supporting exceptions.

2.1 Call Frame Information

One of the key tasks which must be undertaken during the exception handling process is that of unwinding the stack so that the exception may be handled by a handler higher up on the stack. Obviously one may walk the call stack following return address pointers to find all call frames. This is not sufficient for restoring execution, however, as it does not respect register state. Callee-saved registers will not be restored in their normal manner when the execution path of a procedure is interrupted by an exception. It is therefore necessary that the information necessary to restore registers at the time of an unexpected procedure termination (when an exception is thrown from within the procedure) be somehow present at the time of exception throwing/handling.

This is a problem already solved for debugging, as a debugger must do a very similar task when displaying backtraces, allowing the operator to examine the local variables at various levels in the call stack, and son. Therefore, the Call-Frame Information section of the DWARF (Debugging with Attributed Records Format) standard [11] has been adopted for encoding the unwinding information necessary for exception handling with some minor differences, particularly in the area of pointer encoding, which are for the most part documented [20] [1]. It should be noted that the current version of the DWARF standard at the time of this writing is version 4 and most of the information in this paper is drawn from that version. GCC does not in general check which version of DWARF a program was compiled against unless necessary for resolving the layout of a structure or behaviour which conflicts across standards. No checks are made when newer features are used.

Conceptually, what this unwinding information describes is a large table which for every machine instruction in the program text (the rows) describes how to restore the machine state at the previous call frame as if control were to return up the stack from that instruction. The machine state (the columns of this table) is comprised of registers and a Canonical Frame Address (CFA). DWARF allows for an arbitrary number of registers, identified merely by number. It is up to individual ABIs to define a mapping between DWARF register numbers and the hardware registers. The DWARF registers are not required to map to actual hardware registers, for example the return address is encoded as a DWARF register but will not generally correspond to a hardware register. Each cell of this table holds a rule detailing how the contents of the register will be restored for the *previous* call frame. DWARF allows for several types of rules and the curious reader is invited to find them in the DWARF standard [11]. Most registers are restored either from another register or from a memory location accessed at some offset from the CFA. The CFA is an artificial construct (i.e. internal to the DWARF encoding and interpretation) which expresses a canonical address for the call frame on the stack. Most values relevant to the execution of a procedure can therefore be found at some small offset from the CFA. An example (not taken directly from a real program, but modeled after what may be found) of a portion of this table is given in Figure 2.1.

PC (eip)	CFA	ebp	ebx	eax	return addr.
0xf000f000	rsp+16	*(cfa-16)			*(cfa-8)
0xf000f001	rsp+16	*(cfa-16)			*(cfa-8)
0xf000f002	rbp+16	*(cfa-16)		eax=edi	*(cfa-8)
⋮	⋮	⋮	⋮	⋮	⋮
0xf000f00a	rbp+16	*(cfa-16)	*(cfa-24)	eax=edi	*(cfa-8)

Figure 2.1: Example Conceptual Unwinding Table

We note that this table if constructed in its entirety would be absurdly large. It would be considerably larger than the text of the program itself. There are, however, many empty cells and many duplicated entries in columns. Much of the DWARF call frame information standard is essentially a compression technique so that sufficient information may be provided to, at runtime, build parts of the table as needed without the full, prohibitively large, table ever being built or stored. This compression is performed by introducing the concept of Frame Description Entities (FDEs) and DWARF instructions. An FDE corresponds to a logical block of program text (often a procedure although there is no requirement on this) and describes how unwinding may be done from within that procedure. To conserve space, information common to many FDE's is separated into a Common Information Entity (CIE) which holds many of the bookkeeping details. The precise details of the CIE and FDE structures may be found in the DWARF standard. The version of the CIE structure used for `.eh_frame` derives from DWARF versions 2 and 3 and does not include new fields added to the structure in DWARF version 4. A diagrammatic view of CIE and FDE structure is shown in Appendix A. Each FDE contains a series of DWARF instructions. There are two major types of instructions. The first specifies one of the column rules (registers) as from our table above. This rule applies to all cells in that column from the *current location* to the end of the procedure unless a different rule is specified for the same column/register later in the sequence. The *current location* on which these instructions acts begins at the first program text location described by the FDE. The second type of DWARF instruction, location instructions, advances or moves the current location to which the rule instructions apply. In this manner the entire table can be specified in a much smaller form.

2.2 DWARF Expressions

As noted earlier, most of the register rules specify the restoration of a register from another register or from a location on the stack (relative to the CFA). DWARF was not designed for any particular hardware or software platform, however, and there was a very conscious effort to be as flexible as possible. Its designers could not anticipate all ways in which the values of registers were to be restored. Therefore, DWARF version 3 introduced the concept of DWARF expressions (they were present to a much lesser degree in DWARF version 2) which have their own set of instructions. A register may be restored as the result of a DWARF expression. A DWARF expression consists of DWARF expression operations (instructions). These operations are evaluated on a stack-machine Most instructions operate on the top items on the stack. While the DWARF standard does not specify the data format of stack items, GCC implements them as architecture word-sized objects. All of the basic operations necessary for numerical computation are provided: pushing constant values onto the stack, arithmetic operations, bitwise operations, and stack manipulation. In addition, DWARF expressions provide instructions for dereferencing memory addresses and obtaining the values held in registers (DWARF registers calculated as part of the unwind process so far, not necessarily machine registers). This allows registers to be restored from memory locations and registers with additional arithmetic applied. This is a fairly straightforward extension of the simpler register rules provided (with the important difference that memory dereferences may be done on absolute rather than stack-relative addresses). To truly allow register restoration from arbitrarily computed values, however, DWARF expressions include conditional operations and a conditional branch instruction. Due to this extreme flexibility, there is a complete mostly unseen machine capable of arbitrary computation residing in the address space of every GCC-compiled C++ program or program linking C++ code. As an example, if a `char*` string may be found on the stack as the first local variable below the base pointer, the DWARF expression given in Listing 2.1 finds the length of that string and returns it as the result of the expression. A complete explanation of all of the instructions used can be found in the DWARF standard [11].

Listing 2.1: DWARF strlen expression

```
#value at -0x8(%rbp) on stack
DW_OP_breg6 -8
DW_OP_lit0 #initial strlen
DW_OP_swap
DW_OP_dup
#loop begins here
DW_OP_deref_size 1
#branch if top of stack nonzero
DW_OP_bra 3 #forward 3 bytes
DW_OP_skip 8 #skip to the end
#increment the counted length
DW_OP_swap
DW_OP_lit1
DW_OP_plus
DW_OP_over
#add length to char pointer
DW_OP_plus
DW_OP_skip -16 #back 16 bytes
#finally put the character
#count on the top of the stack
#as return value
DW_OP_swap
```

2.3 Exception Handlers

We have observed how the unwinding of stack frames and the accompanying register restoration is performed. It is also necessary to understand how exception handler (catch blocks in C++ terminology) information is encoded. DWARF is designed as a debugging format where the debugger will be in control of how far to unwind the stack. It therefore does not provide any mechanism for encoding what sort of conditions stop the unwinding process. What it does provide is the means for augmentation. Every CIE includes an augmentation string, the contents of which are implementation defined. It is designed to allow a DWARF producer (software creating the DWARF information) to communicate to a DWARF consumer (software reading the

DWARF information) which of a set of previously agreed upon augmentations to the CIE and FDE structures are being used. The augmentations to be used on Linux and x86_64 are well-defined by the respective standards [1] [20]. The defined augmentations allow a language-specific data area (LSDA) and personality routine to be associated with every FDE. Both of these pieces of information are specified as pointers (which may be relative or absolute). When unwinding an FDE, the exception handling process is required to call the personality routine associated with the FDE. The personality routine will interpret the LSDA and determine if a handler for the exception has been found. The actual contents of the LSDA are not defined by any standard. Standardization is viewed as not necessary because the structure that is standardized allows portions of a program written in two different languages with different information that needs to be encoded about exception handlers to coexist and to even pass exceptions between each-other (to a certain degree). The language-specific portions are handled be separate personality routines for the two modules.

The result of this is that the encoding of where exception handlers are located and what type of exceptions they handle is mostly non-standardized and non-documented. What scanty documentation there is not codified in official sources but is only to be found on the gcc mailing lists in posts such as these [36],[31]. These link to an old Hewlett-Packard document [13] which describes the format in fair detail, except it is either wrong or outdated as practical experimentation will show. The best known source of information on the format used by GCC is the assembly code generated by GCC with the flags `-fverbose-asm -dA`. In an ELF binary, the section `.gcc_except_table` contains an array of LSDAs (not ordered in any particular manner, as they are reached from the LSDA pointers in augmented FDEs). Essentially, an LSDA breaks the text region described by the corresponding FDE into call sites. Call sites corresponding to code within a try block (to use C++ terminology) each have a pointer to a chain of C++ typeinfo descriptors. These objects are used by the personality routine to determine whether the thrown exception can be handled by the handler in the current frame. A diagram of LSDA structure can be found in Appendix B.

2.4 Exception Process

Finally, once an understanding of the data driving the exception handling process has been gained, it is important to understand the code path taken during the throwing of an exception. This path is shown in Figure 2.2

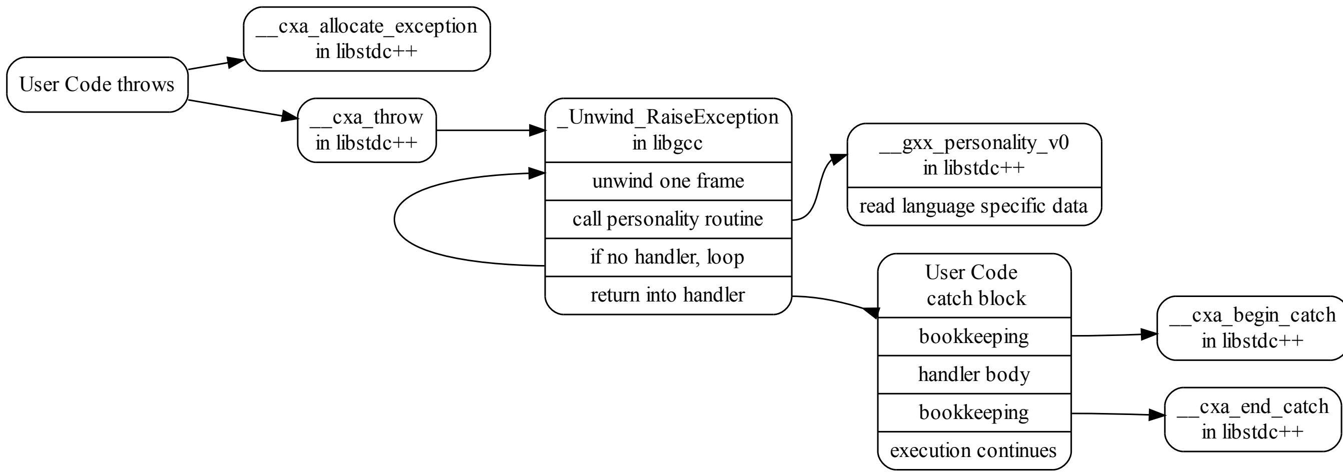


Figure 2.2: C++ Exception Code Flow

The most important facet is that libgcc computes the machine state as a result of the unwinding, directly restores the necessary registers, and then returns into the handler code. We note that at least in current (4.5.2) GCC implementations, this means that at the time execution is first returned to the handler code, the data from which the registers were restored will still be present below the stack pointer until it is overwritten. The handler code is known as the *landing-pad*

With this knowledge, the only barrier to building something interesting with DWARF is the lack of tools to work with it.

Chapter 3

Katana and Dwarfscript

DWARF is chiefly the province of compiler and debugger authors. There are several tools which allow one to examine the DWARF frame information contained within an ELF binary. Known to the authors are readelf, objdump (which uses the same libbfd used by gdb), and dwarfdump [3]. There are, however, no known tools which allow manipulation of DWARF structures at a high level. The author of dwarfdump and libdwarf is known to be working on a DWARF generation tool which will support, among others, a text input mode, but as of this writing he has not made available any code supporting creating DWARF objects from a textual representation. A high-level way of manipulating call frame and exception handler information is essential to examining the security implications of DWARF and demonstrating the consequences of an adversary gaining control of this information.

To bridge this gap, we present **katana**, a tool for ELF and DWARF manipulation, and Dwarfscript, a language for expressing call frame unwinding information and the corresponding exception handler information. Katana as an ELF and DWARF manipulation tool was first developed at Dartmouth College for hotpatching research [5]. The underlying binary manipulation framework was flexible enough that we were able to readily add additional exception-handling related features to it and improve its existing handling of DWARF. Katana provides an easy-to-use tool with a shell-like interface capable of emitting the Dwarfscript representation of an ELF binary. A user may modify the Dwarfscript and then use Katana to assemble it into its binary form which can be reinserted into the executable by Katana. This allows sufficient power to experiment with the consequences of carefully crafted DWARF instructions.

Katana is built on `libelf`[8] and `libdwarf`[3]. `libelf` was chosen instead of the GNU BFD because while the latter offers more features `libelf` allows very simple, bare, access to the underlying ELF structures and thus its use did not constrain the design of Katana, it merely saved labor. `libdwarf` is used when parsing binary DWARF sections. The generation of binary DWARF data is implemented natively in Katana to allow the necessary level of control and because `libdwarf` is more intended for dealing with debugging information and not designed to cope well with generating `.eh_frame`.

3.1 Katana Shell

Programming any automaton such as the DWARF virtual machine requires a reasonable toolchain. To aid our experimentation, it was necessary to develop two languages. The first of these is the Katana shellscript. This turns Katana into a command interpreter/shell in the spirit of Elfsh [18]. The Katana shell has not been designed to be full-featured. It is expanded as necessary for the authors research efforts, and feature requests or suggestions are welcome. It is a simple command interpreter and does not currently support any form of control flow. It supports dynamically typed variables and a small set of commands. The types of variables are as follows:

- strings
- integers
- raw data
- ELF objects
- ELF sections
- arrays

The current command-set is as follows. Some commands have return values which can be assigned to variables by the code `$VARNAME=CMD`

- `load FILENAME`
  Loads data from FILENAME. If it can be interpreted as an ELF object it is. Otherwise it is interpreted as a raw data object. The return value of the command is this object.

- `save OBJECT FILENAME`
  If OBJECT is an ELF object, it is finalized (necessary section headers computed, etc) and saved to FILENAME on disk. If the object is raw data it is simply written byte-for-byte to FILENAME

- `info eh OBJECT`
  Prints information about the low-level exception handling structures for the ELF object OBJECT.

- `hash elf STRING`
  Prints the elf hash (as defined in [34]) for the given STRING.

- `dwarfscript emit [SECTION_NAME] ELF FILENAME` Prints the Dwarfscript representation of the `.eh_frame` section from the given ELF object and saves it to FILENAME. Optionally, a different SECTION_NAME can be used instead of `.eh_frame`.

- `dwarfscript compile FILENAME` Compiles the Dwarfscript file referred to by FILENAME into raw ELF binary sections. Returns an array containing ELF sections for, in order, `.eh_frame`, `.eh_frame_hdr`, `.gcc_except_table`.

3.2 Dwarfscript

The goal of Dwarfscript is to provide a reasonably easy means for a human to quickly make sense of the exception handling data in an ELF binary and to modify it without having to painstakingly rearrange the underlying binary structure with a hex editor. Dwarfscript attempts to encode all information found in the ELF sections `.eh_frame`/`.eh_frame_hdr`, which contain the DWARF unwinding information, and in `.gcc_except_table`, which contains the exception handler information. While it should be perfectly capable of representing DWARF-standard compliant debugging information as well, its focus is on supporting all exception handling information used by GCC.

Dwarfscript is to assembly as binary exception handling data is to machine code. Dwarfscript does not attempt to provide any high-level abstractions over the exception handling data but rather to present it in a form faithful to its binary structure but allowing easy readability and modification. It is a cross between a data-description language (representing the CIE, FDE, and LSDA structures in a textual form) and an assembly language (representing DWARF instructions and expressions as an ASCII-based language). In all cases, care has been taken to follow the structure of the DWARF data as specified in the DWARF standard [11]. The lack of any high-level constructs is deliberate. Dwarfscript allows the manipulator direct control over the data structures involved. The sample of a DWARF expression shown in Listing 2.1 is a valid part of a Dwarfscript file. It is beyond the scope here to show a complete Dwarfscript file but samples may be found in the distribution of Katana and formal grammar is given in Appendix C. Familiarity with the DWARF standard should allow one to understand Dwarfscript, as the goal is to provide a textual representation of what would otherwise be encoded in a compact binary format.

To work with Dwarfscript, one is expected to use Katana to emit the Dwarfscript for the binary which is being modified. One can then edit this Dwarfscript with the changes necessary to fulfill the goal. Katana can then be used to compile the modified Dwarfscript into binary form (the standard DWARF representation) and insert it into the executable binary. To facilitate editing, the Katana distribution contains an Emacs major-mode for Dwarfscript.

The ability to extract information from a binary executable, modify the information, and insert it back into the binary is not a common one and it makes Katana quite powerful for experimenting with binary-level changes to a program.

Chapter 4

Anatomy of a DWARF Trojan

What might an adversary be able to do with control of the exception handling information? In the most naïve case even without complicated DWARF expressions we could redirect the flow to skip a frame when unwinding (if we know the size of the frame on the stack). One of the simplest-possible DWARF expressions allows us to simply set a register to a constant address. Using this means we can redirect any function in our target binary to "return" to any other function in our binary. By manipulating `.gcc_except_table` we can ensure that there is always a landing pad where we would like it.

To demonstrate the power of controlling the exception handling information, we discuss how the ELF binary for a simple program can be modified to yield a shell when an exception is thrown. Our example program merely takes input from `stdin` and prints a response based on the user input. The function of the program is not relevant to our purposes. What is relevant is that if the program receives an input string it is not expecting it throws an exception (the type of the exception is irrelevant here). This program, while perhaps not very interesting, certainly does nothing that would be considered dangerous. An examination of its symbol table reveals that it does not link any of the `exec` family of functions.

We modify the ELF binary for this program in such a way that it will yield a bash shell. An examination of the modified binary will not show any differences in the text or any other section which is interpreted as machine instructions or directly affects the linking of machine instructions. Especially, we do not modify the sections `.text`, `.plt`, `.got`, `.dtors`, `.dynamic`. These sections have long been known as reasonably easy ways to insert backdoors. Modifications are made only to the following ELF sections: `.eh_frame`,

.eh_frame_hdr, .gcc_except_table.

A dynamic linker is built as a DWARF expression which locates the symbol execvpe in libc. An offset is added to this address so that control will be transferred to specific suitable instructions within the function because of the difficulty of controlling parameter passing on x86_64 as discussed in Section 5.1. The specific point that code will be transferred to in the version and build of libc targeted (Arch Linux glibc 2.13-1) is shown in Listing 4.1.

Listing 4.1: Gadget in libc

```
mov     %r12,%rdx
mov     %rbx,%rsi
mov     %r14,%rdi
callq   a4eb0 <execve>
```

The FDE for the function in which the exception is thrown is modified so that one of the registers is set to the result of the dynamic-linking DWARF expression. As seen in Listing 4.1 we are setting up arguments and then calling execve. The call we want to effectively make is execve("/bin/bash", "/bin/bash","-p",NULL,NULL). For alignment reasons, GCC typically leaves extra padding space after .gcc_except_table both in-memory and in the ELF file. Therefore, we have a little extra room to insert some data (which we will know the address of) after the actual LSDA data in .gcc_except_table. We therefore insert the data for these execve parameters here. We then set up the appropriate registers in Dwarfscript as shown in Listing 4.2. Obviously all addresses are specific to where the parameter data was inserted. The DWARF register number of rbx on x86_64 is 3.

There is one significant problem remaining to be solved: we must somehow transfer execution to the place in libc we have picked. We can modify the LSDA data in .gcc_except_table to control where libgcc/libstdc++ thinks handlers are located, but we cannot trivially pretend a handler exists in libc since we do not even know where the library will be loaded (assuming some form of library load ASLR). The solution is to use a classic return-to-libc attack. We take advantage of the fact that the values computed by DWARF will be temporarily placed on the stack in order to be transferred to registers immediately upon return to the handler. We therefore set the stack pointer to just below the location of the computed address in libc on the stack. This does introduce a dependency on particular libgcc/libstdc++ versions for the

Listing 4.2: Dwarfscript `execve` argument setup

```
DW_CFA_val_expression r14
begin EXPRESSION
#set to address of /bin/bash
DW_OP_constu 0x400f2c
end EXPRESSION
DW_CFA_val_expression r3
begin EXPRESSION
#set to address of address of string
#array {"/bin/bash","-p",NULL}
DW_OP_constu 0x400f3a
end EXPRESSION
DW_CFA_val_expression r12
begin EXPRESSION
#set to NULL pointer
DW_OP_constu 0
end EXPRESSION
```

amount of stack space used in handling the exception to be known. One mitigation to this dependency would be to use a DWARF expression to search a small area of the stack for known values which could be used to determine an offset. We set the stack pointer (DWARF register 7 on x86_64) to be a constant offset from the base pointer (DWARF register 6) at the time the exception is thrown as shown in Listing 4.3. We then simply modify the landing pad in the LSDA to point to a return instruction anywhere in the binary being modified. libgcc will transfer control to that return instruction which will return to libc and the process will become a shell.

Listing 4.3: Dwarfscript stack pointer setup

```
DW_CFA_val_expression r7
begin EXPRESSION
DW_OP_breg6 -96
end EXPRESSION
```

4.1 Building a Dynamic Linker in DWARF

Given the general-purpose computational abilities of DWARF expressions it should not be startling that we were able to build a dynamic linker in DWARF. It demonstrates the power of DWARF used in exploits, however and shows another way that address-space layout randomization (ASLR) can be defeated. The only assumptions made are that the `.dynamic` section will not be moved by the loader and that the order that shared libraries are loaded will be the order that they are listed in `.dynamic`. This second assumption is not crucial and the dynamic linker code could easily have been slightly expanded to search for the libc linkmap entry. It is important to note that our DWARF dynamic linker does not simply call the standard linker in ld.so. It traverses the linkmap, hash-table and chain structures directly and thus is not affected by any protections built into the standard linker, such as protection against calling `dlsym` from arbitrary locations. It serves no purpose do discuss the precise details of a dynamic linker here: it is well documented elsewhere [34, 15], and ours is not substantially different in structure. Writing a dynamic linker in DWARF is similar to writing a dynamic linker in assembly except that there are no registers available: there is only a stack instead. A brief outline of the procedure is shown in Listing 4.4 to highlight what sorts of operations DWARF expressions are capable of. Those interested in the actual DWARF code are invited to contact the authors directly, and it will soon be made available in the Katana distribution.

4.2 Combining with Traditional Exploits

What we have concretely demonstrated so far is a trojan technique. The potential of DWARF-based exploits extends beyond this, however. If we have a means of overwriting exception handling data or otherwise making data we control to be interpreted as exception handling data, DWARF can be used in the construction of an exploit delivered at runtime (rather than a trojan).

Hypothetically, a data-injection exploit which may be unable to insert directly executable code could be used to inject DWARF bytecode which will be executed when an exception is thrown. If done, this would aid in circumventing non-executable stacks and heaps and may be an alternative to return-oriented programming requiring less careful piecework construc-

Listing 4.4: Dwarfscript Dynamic Linker Pseudocode

```
dereference DT_PLTGOT+8
top of stack contains a link_map*
while top of stack is not libc linkmap
  top of stack=linkmap->l_next
top of stack=linkmap->l_addr
find DT_HASH, DT_STRTAB, DT_SYMTAB
index into hashtable by "execvpe" hash
while execvpe symbol not found
  compare symbol name and "execvpe"
  if not equal
    symbol found=next in chain
get st_value from symbol
add offset to desired entry point
```

tion and with fewer known detection/mitigation techniques. If this is to be achieved, however, there must be some way to either overwrite `.eh_frame` or to fool the exception handling mechanism into reading data supplied by an attacker instead of the original `.eh_frame`.

There exist many C++ libraries in the wild with `.eh_frame` sections which are loaded read-write. Until 2002 all `.eh_frame` sections were read-write. In 2002 GCC began emitting read-only `.eh_frame` sections unless relocations were necessary for `.eh_frame` [12]. This meant that most PIC code (i.e. libraries) still required writable `.eh_frame`. Modern versions of GCC are now capable of emitting `.eh_frame` sections that do not require relocation even in PIC code, but on up-to-date distributions it is still possible to find libraries with writable `.eh_frame` sections, notably several distributions of the JVM. This is an avenue down which our future work will travel.

It would be even more beneficial to insert an alternate `.eh_frame`, since as time progresses finding binaries with writable exception handling information will become increasingly uncommon, and even at present only a fairly small set of programs will be vulnerable to `.eh_frame` overwriting. `libgcc` locates the `.eh_frame` section through use of the `GNU_EH_FRAME` program header, which points to `.eh_frame_hdr`, which in turn points to `.eh_frame`. Therefore, this program header controls where the DWARF data is read from. Overwriting program headers at runtime is not generally feasible. `libgcc` obtains this program header through the function `dl_iterate_phdr` which is located in `libld`. Again, not an easy target. `libgcc` caches program headers, however, in the variable `frame_hdr_cache`, which is static to its compilation unit. As this is a static variable its symbol is not exported, and thus its precise location will depend on the build. While `libgcc` exports no symbols, after an exception returns there will be text addresses within `libgcc` on the invalid portion of the stack, some up to nearly 1k below the stack pointer. It is likely possible to correlate these addresses with the data address of the header cache. We have confirmed that overwriting this header cache can allow the interpretation of arbitrary data as `.eh_frame`. Fully weaponizing this into a workable exploit remains as future work.

Chapter 5

Limitations and Workarounds

There are several limits which anyone attempting to use the DWARF virtual machine for arbitrary computation will encounter.

5.1 Registers and Parameter Passing

First, it is important to note that not all machine registers may be restored during stack unwinding. The hardware ABI will define some set of registers which are callee-saved and some set which are not guaranteed to be saved. It is reasonable to assume that the unwinding implementation will restore all callee-saved registers as specified in the DWARF instructions. We have determined, both empirically and through examination of the GCC source code, that at least on the x86_64 platform, the values assigned to certain registers through DWARF will be ignored when restoring the call frame to return into. Some of the non callee-saved registers are used for specific exception-handling purposes (i.e. on x86_64 , rdx is used to store an identifier for the type of exception thrown) and some appear simply not to be restored to any value. This presents a problem on architectures like x86_64 where registers are used for passing function parameters. The authors of a backdoor would like to be able to return execution to the beginning of some interesting function (such as `execv` in libc) and pass it crafted arguments. The x86_64 registers used for argument passing (rdi, rsi, rdx, rcx, r8, r9) cannot be restored, however, making this impossible. It therefore becomes necessary to return to a point inside the target function which makes use of registers that can be controlled. If the target function resides in a library this can make it difficult for an exploit to be portable across multiple versions or even builds of the library. This problem can be mitigated to some degree if it is possible to find a suitable landing pad in the binary being modified which calls a function pointer. If the address of the library function can be called as the function pointer then dependence on the precise build of library functions will be reduced. This problem is also lessened on architectures such as 32-bit x86 which primarily use the stack rather than registers for argument passing. Another workaround for this difficulty is to code the exploit for several versions/builds. As long as it is possible to make a value appear on the target's stack (this can be in innocuous means, such as expected user input) this value can be searched for the DWARF program and used as a parameter, allowing the code path taken to be adjusted when the attack is triggered rather than merely at the time of injection.

5.2 No Side Effects

Through control of the stack and base pointers, a DWARF program can to some degree control the contents of the stack when execution is resumed. DWARF instructions/expressions do not, however, have the ability to directly modify memory or push anything onto the stack therefore it can be difficult to make the code at the landing pad access values on the stack that were calculated by the DWARF expression. One workaround to this is to exploit the fact that values computed for machine registers will still be in memory (at least until overwritten by new stack frames) as they are computed in memory and then transferred into the correct registers. The DWARF program can set the stack or base pointer to point to the correct region of memory. There are two limitations to this technique. The first is that the appropriate location where the computed values will be found is highly dependent on the precise stack layout and thus varies between versions and builds of libgcc and libstdc++. The second limitation is that there is of course limited contiguous stack space that can be controlled, limited by the number of restored registers and how they are laid out in memory.

5.3 DWARF Machine Implementation

Obviously the implementation of the DWARF virtual machine has some effect upon what sort of computations can be performed. The current (gcc 4.5.2) implementation in libgcc allows the DWARF stack to grow only to a size of 64 words [12]. The DWARF standard does not specify the maximum size of the stack and there does not appear to be a reasoned processes behind this number, rather it appears to have been arbitrarily chosen as a size which should be large enough for any DWARF program gcc would expect to be necessary. While this limit should be kept in mind when writing a DWARF program it does not seriously hamper the creation of interesting DWARF programs. As discussed in Section 4.1, a dynamic linker can be programmed in DWARF using only 16 words on the stack.

5.4 Limited `.eh_frame` space

When modifying an ELF binary, we cannot count on the presence of full relocation information as GCC/`ld` does not by default emit relocatable ELF objects. Therefore, we definitely cannot count on being able to expand `.eh_frame` and we would like to avoid moving it as well. Therefore, we must be careful that DWARF programs and other modifications to FDE, CIE, and LSDA structure do not require expanding the size of `.eh_frame`. This limitation can be fairly easily overcome, however. GCC does not attempt to perform any static analysis to determine whether the call frame for a given function will ever be unwound during exception handling. `.eh_frame` will even be generated for C compilation units despite the fact that C does not support exception handling. The reason for this is that it allows exceptions to propagate seamlessly across areas of code which do not know how to deal with them. Human analysis of the program being modified, however, should yield insight into finding FDEs corresponding to functions which will never need to be unwound. In Dwarfscript, these FDEs can simply be removed from the script file, making available more room for lengthy DWARF expressions. The dynamic linker referenced earlier and discussed in Section 4.1 requires less than 200 bytes of space for its instructions.

5.5 Debugging

There are presently no tools available to debug DWARF programs. Rudimentary debugging can be achieved by stepping through the execution of the DWARF virtual machine in libgcc with a debugger. DWARF debugging support is a planned feature for Katana.

Chapter 6

Conclusion

We have demonstrated how the hitherto largely unexplored DWARF-format exception handling information used on a wide-variety of Unix and Unix-like platforms can be used to control the flow of execution. This has several advantages over traditional backdoors and over return-oriented-programming. Advantages of our technique include the following

- Turing-complete environment.
- ASLR defeating.
- Less likely to be detected by traditional executable-content scanners.
- Built-in trigger mechanism (the attack can lie dormant until an exception is thrown).
- Fewer carefully chained gadgets required in the target program than in return-oriented-programming therefore less analysis and time may be necessary to develop an attack.
- Does not rely on bugs. Our DWARF programs leverage existing mechanisms as an extension of their intended purpose and do not rely on implementation bugs and outright security holes but on deliberately made behavior and mechanisms.

Notably, DWARF expressions can read registers and process memory, and, as we have shown, they are suitable for writing a fully functional dynamic linker.

We stress the security risks associated with powerful computational environments added in unexpected places. While being a sterling example of extensible software engineering and introducing a conceptually graceful method of handling complex datastructures of previously unprecedented complexity, this DWARF subsystem also far exceeds expectations of most developers and defenders regarding the computations it is capable of performing.

This may lead defenders to underestimate the risks posed by such environments, and miss a number of possible attack vectors.

Finally, we release Katana as a tool to painlessly create and experiment with the sort of crafted DWARF programs we have discussed, so that interested researchers can further explore the relevant attack surface.

6.1 Availability

Katana is available under the GNU General Public License and may be found at `http://katana.nongnu.org/`.

6.2 Acknowledgments

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0524695. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We owe a debt of thanks to projects and people who have inspired us or have helped us along the way. These include (but are not necessarily limited to) Julien Vanegue (the ERESI project [35]) , The Grugq (*Cheating the ELF* [33]), Nergal (*The advanced return-into-lib(c) exploits: PaX case study* [21]), Skape (*LOCREATE* [28]), and Tim Yardley (for helpful advice). We are of course grateful to Professor Sean Smith and the Dartmouth College Trust Lab for continuing support.

Bibliography

[1] Linux standard base core specification 4.0. http://refspecs.linux-foundation.org/LSB_4.0.0/LSB-Core-generic/.

[2] The month of kernel bugs archive. http://projects.info-pull.com/mokb/, Nov. 2006.

[3] Anderson, D. Libdwarf and dwarfdump. http://reality.sgiweb.org/davea/dwarf.html, Jan 2011.

[4] anonymous. Once upon a free(). *Phrack Magazine 57*, 9 (Aug 2001).

[5] Bratus, S., Oakley, J., Ramaswamy, A., Smith, S., and Locasto, M. Katana: Towards patching as a runtime part of the compiler-linker-loader toolchain. *International Journal of Secure Software Engineering (IJSSE) 1*, 3 (2010), 1–17.

[6] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: generalizing return-oriented programming to RISC. In *ACM Conference on Computer and Communications Security* (2008), pp. 27–38.

[7] Cesare, S. Shared library redirection via ELF PLT infection. *Phrack Magazine 56*, 7 (May 2000).

[8] Drepper, U., McGrath, R., and Machata, P. libelf. https://fedorahosted.org/elfutils/.

[9] Dullien, T., Kornau, T., and Weinmann, R.-P. A framework for automated architecture-independent gadget search. W00T '10: 4th USENIX Workshop on Offensive Technologies.

[10] Durden, T. Bypassing pax aslr protection. *Phrack Magazine 59*, 9 (Jul 2002).

[11] DWARF Debugging Information Format Committee. DWARF debugging information format version 4. http://dwarfstd.org/, June 2010.

[12] Free Software Foundation. GCC 4.5.2 source code. http://gcc.gnu.org/releases.html, December 2010.

[13] Hewlett Packard Corporation. aC++ A.01.15. http://www.codesourcery.com/public/cxx-abi/exceptions.pdf, Dec 2001.

[14] Hund, R., Holz, T., and Freiling, F. Return-oriented rootks: Bypassing kernel code integrity protection mechanisms. USENIX Association.

[15] Levine, J. R. *Linkers and Loaders*. Morgan-Kauffman, 1999.

[16] LLVM Team. DwarfException.cpp. `http://llvm.org/svn/llvm-project/llvm/trunk/lib/CodeGen/AsmPrinter/DwarfException.cpp`. Evidence of llvm support of DWARF and gcc_except_table format.

[17] lzik. Abusing .ctors and .dtors for fun 'n profit.

[18] mayhem@devhell.org. Embedded elf debugging : the middle head of cerberus. *Phrack Magazine 11*, 61 (2003).

[19] Meer, H. Memory corruption attacks: The (almost) complete history. Black Hat.

[20] Michael Matz, Jan Hubi ka, A. J. M. M. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, draft version 0.99.5 ed., September 2010.

[21] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine 58*, 4 (Dec 2001).

[22] O'Neill, R. Modern day elf runtime infection via got poisoning.

[23] Openwall Project. Linux kernel patch from the openwall project. `http://www.openwall.com/linux/`.

[24] Richarte, G. About exploits writing. `http://corelabs.coresecurity.com/index.php?module=Wiki&action=attachment&type=area&page=Vulnerability_Research&file=publication%2FAbout_Exploits_Writing%2F2002.gera.About_Exploits_Writing.pdf`, 2002.

[25] Richarte, G., and Riq. Advances in format string exploitation. *Phrack Magazine 59*, 7 (July 2002).

[26] Rivas, J. M. B. Overwriting the .dtors section.

[27] Shacham, H. 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* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[28] skape. Locreate: an anagram for relocate. *Uninformed 6* (Jan 2007).

[29] SolarDesigner. Getting around non-executable stack (and fix). Bugtraq mailing list, `http://www.securityfocus.com/archive/1/7480`, Aug. 1997.

[30] Stewart, J. ollybone: Semi-automatic unpacking on ia-32.

[31] Taylor, I. L. Re: Elf-section .gcc_except_table. `http://gcc.gnu.org/ml/gcc-help/2010-09/msg00116.html`, September 2010. gcc-help@gcc.org Mailing List.

[32] Team, P. Pax project. `http://pax.grsecurity.net/`.

[33] the grugq. Cheating the elf: subversive dynamic linking to libraries.

[34] TIS Committee. *elf*, 1995.

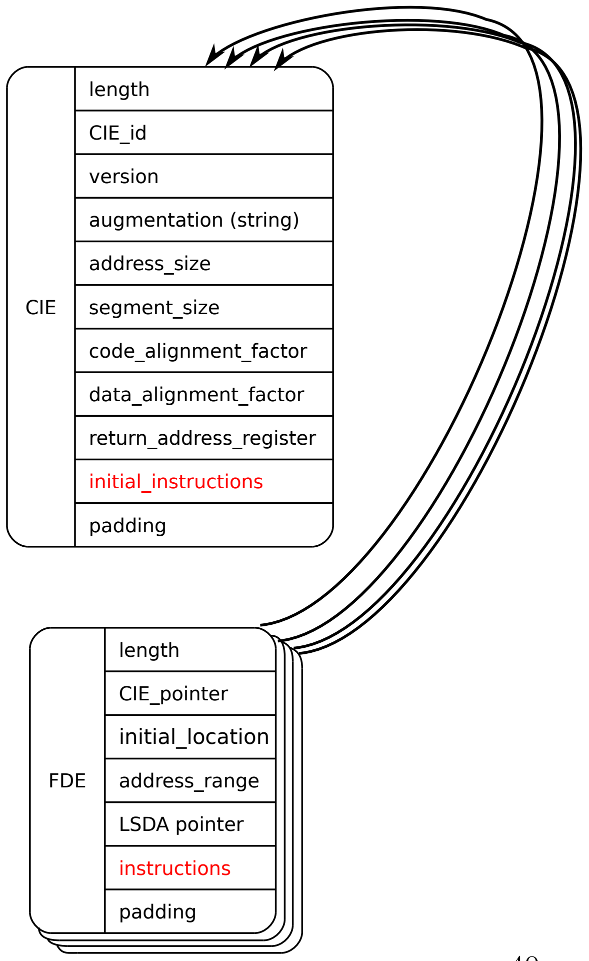
[35] Vanegue, J., Medeiros, J. A. d., Bisolfati, E., Desnos, A., Figueredo, T., Garnier, T., Lesniak, R., Palencia, J., Roy, S., Soudan, S., Woloszyn, M., and Zabrocki, A. The ERESI reverse engineering software interface. `http://www.eresi-project.org/`, 2009.

[36] Various. gcc@gcc.org mailing list: Exception handling description. `http://gcc.gnu.org/ml/gcc/2009-05/msg00390.html`, May 2009.

[37] Wojtczuk, R. Deafeating solar designer's non-executable stack patch. `http://insecure.org/sploits/non-executable.stack.problems.html`, January 1998.

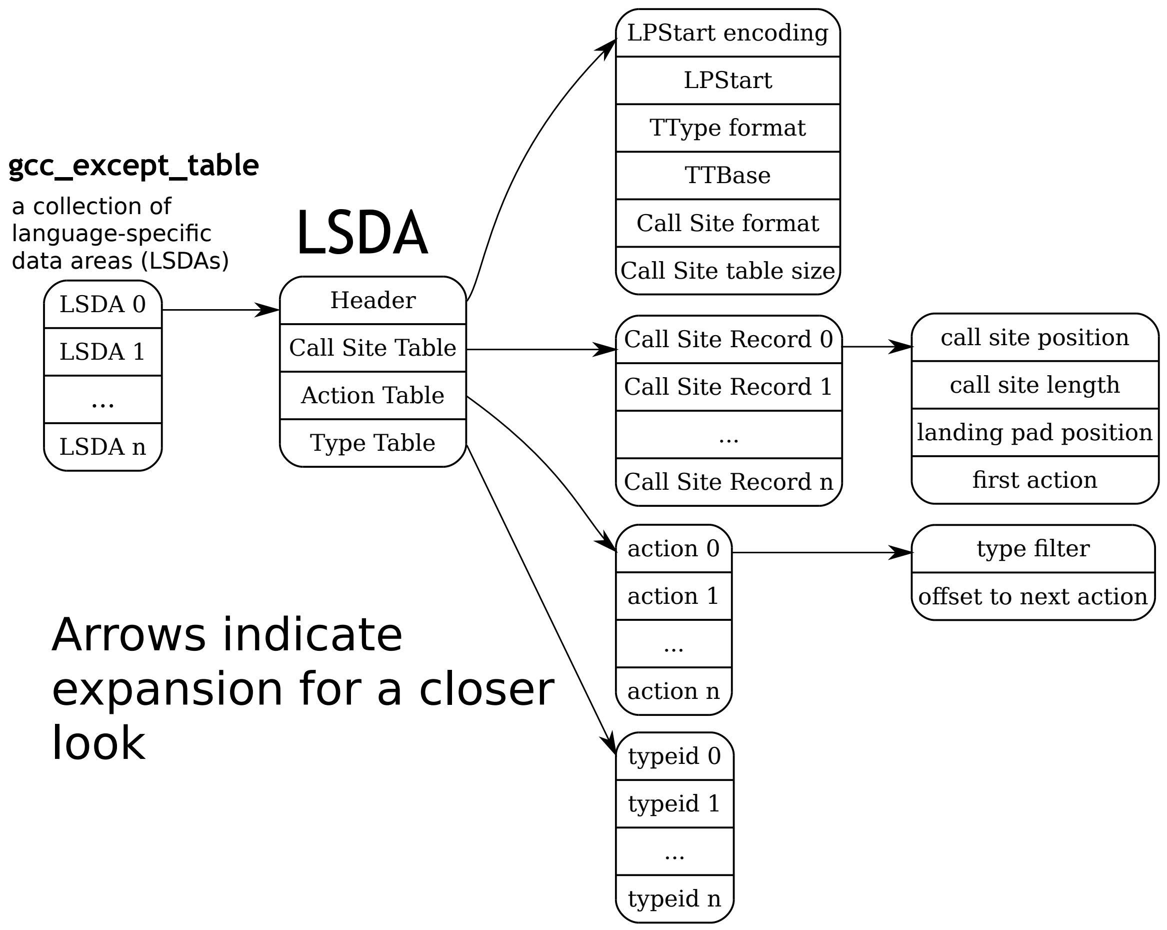
Appendix A

CIE and FDE Structure Inside `.eh_frame`



Appendix B

`.gcc_except_table` Layout



Appendix C

Dwarfscript Grammar

<dwarfscript> → <top_property_stmt_list> <section_list>

<section_list> → <section_list> <fde_section>
| <section_list> <cie_section>
| <section_list> <lsda_section>
| ϵ

<fde_section> → **begin fde** <fde_property_stmt_list> <instruction_section>
<fde_property_stmt_list> **end fde**

<cie_section> → **begin cie** <cie_property_stmt_list> <instruction_section>
<cie_property_stmt_list> **end cie**

<instruction_section> → **begin instructions** <instruction_stmt_list>
end instructions

<expression_section> → **begin dwarf_expr** <expr_stmt_list>
end dwarf_expr

<lsda_section> → **begin lsda** <lsda_part_list> **end lsda**

<call_site_section> → **begin call_site** <call_site_property_stmt_list>
end call_site

<action_section> → **begin action** <action_property_stmt_list>
end action

<top_property_stmt_list> → <top_property_stmt_list> <top_property_stmt>

<fde_property_stmt_list> → <fde_property_stmt_list> <fde_property_stmt>

<cie_property_stmt_list> → <cie_property_stmt_list> <cie_property_stmt>

<instruction_stmt_list> → <instruction_stmt_list> <instruction_stmt>
| ϵ

<expr_stmt_list> → <expr_stmt_list> <expr_stmt>
| <expr_stmt_list> <label>
| ϵ

<lsda_part_list> → <lsda_part_list> <lsda_part>
| ϵ

<call_site_property_stmt_list> → <call_site_property_stmt_list>
<call_site_property_stmt>
| ϵ

<action_property_stmt_list> → <action_property_stmt_list> <action_property_stmt>
|ϵ

<top_property_stmt> → <section_type_prop>
|<section_location_prop>
|<eh_hdr_location_prop>
|<except_table_addr_prop>

<cie_property_stmt> → <index_prop>
|<length_prop>
|<version_prop>
| <augmentation_prop>

|<fde_ptr_enc_prop>
| <fde_lsda_ptr_enc_prop>
|<personality_ptr_enc_prop>
| <personality_prop>
|<address_size_prop>
| <segment_size_prop>
|<data_align_prop>
| <code_align_prop>
| <return_addr_rule_prop>

fde_property_stmt →<index_prop>
| <length_prop>
| <cie_index_prop>
| <initial_location_prop>
| <address_range_prop>
| <lsda_idx_prop>

<lsda_part> → <lsda_property_stmt>
| <call_site_section>
| <action_section>

<lsda_property_stmt> → <lpstart_prop>
| <typeinfo_enc_prop>
| <typeinfo_prop>

<call_site_property_stmt> → <position_prop>
| <length_prop>
| <landing_pad_prop>
| <has_action_prop>
| <first_action_prop>

<action_property_stmt> → <type_idx_prop>
|<next_prop>

<index_prop> → **index :** <nonneg_int_lit>

<length_prop> → **length :** <nonneg_int_lit>

<cie_index_prop> → **cie_index :** <nonneg_int_lit>

<initial_location_prop> → **initial_location :** <nonneg_int_lit>

<address_range_prop> → **address_range :** <nonneg_int_lit>

<version_prop> → **version :** <nonneg_int_lit>

<fde_ptr_enc_prop> → **fde_ptr_enc :** <dw_pe_lit>

<fde_lsda_ptr_enc_prop> → **fde_lsda_ptr_enc :** <dw_pe_lit>

<personality_ptr_enc_prop> → **personality_ptr_enc :** <dw_pe_lit>

<personality_prop> → **personality :** <nonneg_int_lit>

<address_size_prop> → **address_size :** <nonneg_int_lit>

<segment_size_prop> → **segment_size :** <nonneg_int_lit>

<data_align_prop> → **data_align :** <int_lit>

<code_align_prop> → **code_align :** <nonneg_int_lit>

<return_addr_rule_prop> → **ret_addr_rule :** <nonneg_int_lit>

<section_type_prop> → **section_type :** <string_lit>

<section_location_prop> → **section_loc :** <nonneg_int_lit>

<eh_hdr_location_prop> → **eh_hdr_loc :** <nonneg_int_lit>

<except_table_addr_prop> → **except_table_addr :** <nonneg_int_lit>

<lpstart_prop> → **lpstart :** <nonneg_int_lit>

<typeinfo_enc_prop> → **typeinfo_enc :** <dw_pe_lit>

<typeinfo_prop> → **typeinfo :** <nonneg_int_lit>

<position_prop> → **position :** <nonneg_int_lit>

<landing_pad_prop> → **landing_pad :** <nonneg_int_lit>

<has_action_prop> → **has_action :** <bool_lit>

<first_action_prop> → **first_action :** <nonneg_int_lit>

<type_idx_prop> → **type_idx :** <nonneg_int_lit>

<next_prop> → **next :** <nonneg_int_lit>
| **next : none**

<lsda_idx_prop> → **lsda_idx :** <nonneg_int_lit>
<int_lit> → <digits>
| **-** <digits>

<digits> → /[**0-9**]+/
<register_lit> → **r** <digits>

<nonneg_int_lit> → <digits>

<bool_lit> → **true**
|**false**

<dw_pe_lit> → <dw_pe_lit> **,** <dw_pe_lit>
| **DW_EH_PE_absptr**
| **DW_EH_PE_uleb128**
| **DW_EH_PE_udata2**
| **DW_EH_PE_udata4**
| **DW_EH_PE_udata8**
| **DW_EH_PE_sleb128**
| **DW_EH_PE_sdata2**
| **DW_EH_PE_sdata4**
| **DW_EH_PE_sdata8**
| **DW_EH_PE_pcrel**
| **DW_EH_PE_textrel**
| **DW_EH_PE_datarel**
| **DW_EH_PE_funcrel**
| **DW_EH_PE_aligned**
| **DW_EH_PE_indirect**
| **DW_EH_PE_omit**

<instruction_stmt> → <dw_cfa_set_loc>
| <dw_cfa_advance_loc>
| <dw_cfa_advance_loc1>
| <dw_cfa_advance_loc2>
| <dw_cfa_advance_loc4>
| <dw_cfa_offset>
| <dw_cfa_offset_extended>
| <dw_cfa_offset_extended_sf>
| <dw_cfa_restore>
| <dw_cfa_restore_extended>
| **DW_CFA_nop**
| <dw_cfa_undefined>
| <dw_cfa_same_value>
| <dw_cfa_register>
| <dw_cfa_remember_state>
| <dw_cfa_restore_state>

| <dw_cfa_def_cfa>
| <dw_cfa_def_cfa_sf>
| <dw_cfa_def_cfa_register>
| <dw_cfa_def_cfa_offset>
| <dw_cfa_def_cfa_offset_sf>
| <dw_cfa_def_cfa_expression>
| <dw_cfa_expression>
| <dw_cfa_val_offset>
| <dw_cfa_val_offset_sf>
| <dw_cfa_val_expression>

<dw_cfa_set_loc> → **DW_CFA_set_loc** <nonneg_int_lit>

<dw_cfa_advance_loc> → **DW_CFA_advance_loc** <nonneg_int_lit>

<dw_cfa_advance_loc1> → **DW_CFA_advance_loc1** <nonneg_int_lit>

<dw_cfa_advance_loc2> → **DW_CFA_advance_loc2** <nonneg_int_lit>

<dw_cfa_advance_loc4> → **DW_CFA_advance_loc4** <nonneg_int_lit>

<dw_cfa_offset> → **DW_CFA_offset** <register_lit><nonneg_int_lit>

<dw_cfa_offset_extended> → **DW_CFA_offset_extended** <register_lit>
<nonneg_int_lit>

<dw_cfa_offset_extended_sf> → **DW_CFA_offset_extended_sf**
<register_lit> <int_lit>

<dw_cfa_val_offset> → **DW_CFA_val_offset** <register_lit> <nonneg_int_lit>

<dw_cfa_val_offset_sf> → **DW_CFA_val_offset_sf** <register_lit>
<int_lit>

<dw_cfa_restore> → **DW_CFA_restore** <register_lit>

<dw_cfa_restore_extended> → **DW_CFA_restore_extended** <register_lit>

<dw_cfa_undefined> → **DW_CFA_undefined** <register_lit>

<dw_cfa_same_value> → **DW_CFA_same_value** <register_lit>

<dw_cfa_register> → **DW_CFA_register** <register_lit> <register_lit>

<dw_cfa_remember_state> → **DW_CFA_remember_state**

<dw_cfa_restore_state> → **DW_CFA_restore_state**
<dw_cfa_def_cfa> → **DW_CFA_def_cfa** <register_lit> <nonneg_int_lit>

<dw_cfa_def_cfa_sf> → **DW_CFA_def_cfa_sf** <register_lit> <int_lit>

<dw_cfa_def_cfa_register> → **DW_CFA_def_cfa_register** <register_lit>

<dw_cfa_def_cfa_offset> → **DW_CFA_def_cfa_offset** <nonneg_int_lit>

<dw_cfa_def_cfa_offset_sf> → **DW_CFA_def_cfa_offset_sf** <int_lit>

<dw_cfa_def_cfa_expression> → **DW_CFA_def_cfa_expression**
<expression_section>

<dw_cfa_expression> → **DW_CFA_expression** <register_lit>
<expression_section>

<dw_cfa_val_expression> → **DW_CFA_val_expression** <register_lit>
<expression_section>

<expr_stmt> → <dw_op_addr>
| **DW_OP_deref**
| <dw_op_const1u>
| <dw_op_const1s>
| <dw_op_const2u>
| <dw_op_const2s>
| <dw_op_const4u>
| <dw_op_const4s>
| <dw_op_const8u>
| <dw_op_const8s>
| <dw_op_constu>
| <dw_op_consts>
| **DW_OP_dup**
| **DW_OP_drop**
| **DW_OP_over**
| <dw_op_pick>
| **DW_OP_swap**
| **DW_OP_rot**
| **DW_OP_xderef**
| **DW_OP_abs**
| **DW_OP_and**
| **DW_OP_div**
| **DW_OP_minus**
| **DW_OP_mod**
| **DW_OP_mul**
| **DW_OP_neg**
| **DW_OP_not**
| **DW_OP_or**
| **DW_OP_plus**
| <dw_op_plus_uconst>
| **DW_OP_shl**
| **DW_OP_shr**
| **DW_OP_shra**
| **DW_OP_xor**
| <dw_op_skip>
| <dw_op_bra>
| **DW_OP_eq**
| **DW_OP_ge**

| **DW_OP_gt**
| **DW_OP_le**
| **DW_OP_lt**
| **DW_OP_ne**
| <dw_op_litn>
| <dw_op_regn>
| <dw_op_bregn>
| <dw_op_regx>
| <dw_op_bregx>
| <dw_op_deref_size>
| <dw_op_xderef_size>
| **DW_OP_nop**

<label> → <identifier> **:**

<identifier> → /**[a-zA-Z_]**[**a-zA-Z0-9_**]*/

<dw_op_addr> → **DW_OP_addr** <nonneg_int_lit>

<dw_op_const1u> → **DW_OP_const1u** <nonneg_int_lit>

<dw_op_const1s> → **DW_OP_const1s** <int_lit>

<dw_op_const2u> → **DW_OP_const2u** <nonneg_int_lit>

<dw_op_const2s> → **DW_OP_const2s** <int_lit>

<dw_op_const4u> → **DW_OP_const4u** <nonneg_int_lit>

<dw_op_const4s> → **DW_OP_const4s** <int_lit>

<dw_op_const8u> → **DW_OP_const8u** <nonneg_int_lit>

<dw_op_const8s> → **DW_OP_const8s** <int_lit>

<dw_op_constu> → **DW_OP_constu** <nonneg_int_lit>

<dw_op_consts> → **DW_OP_consts** <int_lit>

<dw_op_pick> → **DW_OP_pick** <nonneg_int_lit>

<dw_op_xderef> → **DW_OP_xderef**

<dw_op_plus_uconst> → **DW_OP_plus_uconst**

<dw_op_skip> → **DW_OP_skip** <int_lit>
| **DW_OP_skip** <identifier>

<dw_op_bra> → **DW_OP_bra** <int_lit>
| **DW_OP_bra** <identifier>

<dw_op_litn> → **DW_OP_lit0**
| **DW_OP_lit1**
| **DW_OP_lit2**
| **DW_OP_lit3**
| **DW_OP_lit4**
| **DW_OP_lit5**
| **DW_OP_lit6**
| **DW_OP_lit7**
| **DW_OP_lit8**
| **DW_OP_lit9**
| **DW_OP_lit10**
| **DW_OP_lit11**
| **DW_OP_lit12**
| **DW_OP_lit13**
| **DW_OP_lit14**
| **DW_OP_lit15**
| **DW_OP_lit16**
| **DW_OP_lit17**
| **DW_OP_lit18**

| **DW_OP_lit19**
| **DW_OP_lit20**
| **DW_OP_lit21**
| **DW_OP_lit22**
| **DW_OP_lit23**
| **DW_OP_lit24**
| **DW_OP_lit25**
| **DW_OP_lit26**
| **DW_OP_lit27**
| **DW_OP_lit28**
| **DW_OP_lit29**
| **DW_OP_lit30**
| **DW_OP_lit31**

<dw_op_regn> → **DW_OP_reg0**
| **DW_OP_reg1**
| **DW_OP_reg2**
| **DW_OP_reg3**
| **DW_OP_reg4**
| **DW_OP_reg5**
| **DW_OP_reg6**
| **DW_OP_reg7**
| **DW_OP_reg8**
| **DW_OP_reg9**
| **DW_OP_reg10**
| **DW_OP_reg11**
| **DW_OP_reg12**
| **DW_OP_reg13**
| **DW_OP_reg14**
| **DW_OP_reg15**
| **DW_OP_reg16**
| **DW_OP_reg17**
| **DW_OP_reg18**
| **DW_OP_reg19**
| **DW_OP_reg20**
| **DW_OP_reg21**
| **DW_OP_reg22**

| **DW_OP_reg23**
| **DW_OP_reg24**
| **DW_OP_reg25**
| **DW_OP_reg26**
| **DW_OP_reg27**
| **DW_OP_reg28**
| **DW_OP_reg29**
| **DW_OP_reg30**
| **DW_OP_reg31**

<dw_op_bregn> → **DW_OP_breg0** <int_lit>
| **DW_OP_breg1** <int_lit>
| **DW_OP_breg2** <int_lit>
| **DW_OP_breg3** <int_lit>
| **DW_OP_breg4** <int_lit>
| **DW_OP_breg5** <int_lit>
| **DW_OP_breg6** <int_lit>
| **DW_OP_breg7** <int_lit>
| **DW_OP_breg8** <int_lit>
| **DW_OP_breg9** <int_lit>
| **DW_OP_breg10** <int_lit>
| **DW_OP_breg11** <int_lit>
| **DW_OP_breg12** <int_lit>
| **DW_OP_breg13** <int_lit>
| **DW_OP_breg14** <int_lit>
| **DW_OP_breg15** <int_lit>
| **DW_OP_breg16** <int_lit>
| **DW_OP_breg17** <int_lit>
| **DW_OP_breg18** <int_lit>
| **DW_OP_breg19** <int_lit>
| **DW_OP_breg20** <int_lit>
| **DW_OP_breg21** <int_lit>
| **DW_OP_breg22** <int_lit>
| **DW_OP_breg23** <int_lit>
| **DW_OP_breg24** <int_lit>
| **DW_OP_breg25** <int_lit>
| **DW_OP_breg26** <int_lit>

| **DW_OP_breg27** <int_lit>
| **DW_OP_breg28** <int_lit>
| **DW_OP_breg29** <int_lit>
| **DW_OP_breg30** <int_lit>
| **DW_OP_breg31** <int_lit>

<dw_op_regx> → **DW_OP_regx** <nonneg_int_lit>

<dw_op_bregx> → **DW_OP_bregx** <nonneg_int_lit> <int_lit>

<dw_op_deref_size> → **DW_OP_deref_size** <nonneg_int_lit>

<dw_op_xderef_size> → **DW_OP_xderef_size** <nonneg_int_lit>