6-2-2011

Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques Without Native Executable Code

James M.H. Oakley
Dartmouth College

Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques Without Native Executable Code

Dartmouth Computer Science Technical Report TR2011-688

James Oakley
Advised by Sergey Bratus

Computer Science Department.
Dartmouth College
Hanover, New Hampshire
james.oakley.11@alum.dartmouth.org

June 2, 2011

An earlier report of these results appeared as TR2011-680.

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.

Contents

Abstract **1**

1 Introduction **4**
1.1 Computational models. 5
1.2 Attacking Exceptions. 6
1.3 Contributions 6

2 DWARF and Exception Handling Details **8**
2.1 Environment 8
2.2 Call Frame Information 9
2.3 DWARF Expressions 11
2.4 Exception Handlers 12
2.5 Exception Process 14

3 Katana and Dwarfscript **16**
3.1 Katana Shell 17
3.2 Dwarfscript 18

4 Malicious DWARFs **20**
4.1 A Trojan 20
4.2 Building a Dynamic Linker in DWARF 23
4.3 Detection By Antivirus Software 23
4.4 Combining with Traditional Exploits 25
4.5 Limitations and Workarounds 26
4.5.1 Registers and Parameter Passing 26
4.5.2 No Side Effects 27
4.5.3 DWARF Machine Implementation 28
4.5.4 Limited `.eh_frame` space 28

4.5.5 Debugging 29
4.6 Defenses 29

5 History and Related Work **30**
5.1 Security of Exception Handling 30
5.2 Historical Exploitation 31
5.3 Auxiliary Computation 32
5.4 Where Does DWARF Fit In? 33

6 Conclusion **34**
6.1 Availability 35
6.2 Acknowledgments 35

Bibliography **37**

A CIE and FDE Structure Inside `.eh_frame` **40**

B `.gcc_except_table` Layout **41**

C Dwarfscript Grammar **42**

Chapter 1

Introduction

When a program is executed on a computer system, that system must be able to transform the program on disk into a running process image. The standard format used on Linux and most other UNIX-like systems for representing both programs on disk and process images is the ELF (Executable and Linking Format) [43]. An ELF structure consists of various header information and an array of named sections, each section containing the data necessary for some specific part of the process or process lifecycle (e.g. program text, writable data, symbol table, dynamic linking information, and so on). For languages, such as C++, supporting exceptions, the process image and surrounding mechanisms must include a facility for providing exception handling. While on Windows a process stores this information on the stack [28], on Linux and other systems exception handling information is stored in ELF sections using a standardized format called DWARF (Debugging with Attributed Records Format) [**?**]. We show that if an attacker can gain control of the DWARF data he can perform sophisticated computations unhindered by standard protections such as non-executable stacks and ASLR. This is useful in at least two attack scenarios:

1. The adversary creates a trojan by modifying the DWARF sections of the executable before it is run to create a backdoor. This is less likely to be noticed than many other trojan techniques because no executable code is modified and for further reasons discussed later.

2. The adversary exploits a vulnerability ability allowing data overwrite but not code execution. He uses this exploit to overwrite the DWARF portion of the executable (or the information used to locate the DWARF

data) and causes an exploit to be thrown, allowing the crafted data to do the rest of the work.

1.1 Computational models.

Let us consider the mechanics of these scenarios in terms of computational models. Historically, exploitation has focused on the main computation, i.e. on the program text. When many people think of compiling and executing a C or C++ program, they think of the process of transforming the source code into machine code and then executing the machine code on the processor. Traditional exploitation techniques focus either on introducing new code to be executed directly (e.g. standard Aleph One style buffer overflows with shellcode payloads [2]) or on manipulating the flow of execution of existing code (e.g. return to libc [29, 40] or more recently return-oriented programming (ROP) [38]). In ELF terms, attention is generally focused on the `.text` sections of the main program and any shared libraries it loads. While control of the main computation is nearly always the desired goal of exploitation, that control can be achieved in ways other than focusing solely on that computation.

The main computation is by no means the only computation. Besides the well-understood program text and data sections, a modern ELF file may contain over 30 auxiliary sections which contain data and "code" controlling various stages of the process life-cycle, including loading, dynamic linking of required library functions, relocation of symbols, process tear down, and — the focus of this paper — exception-handling. An automaton (not necessarily a Turing machine) performs each of these tasks, using the relevant ELF sections as input. Therefore, the input to each of these automata is a program of sorts, albeit not a directly hardware-executable one. As demonstrated by works such as Locreate [39], these automata can be used to accomplish potentially surprising results that influence the main program.

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 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.

1.2 Attacking Exceptions.

There are several reasons why attacking exception handling is attractive. A backdoor triggered only by exception handling may be difficult to detect. Aside from being unexpected (since this novelty will of course eventually wear off), the backdoor creates a very low risk of disrupting normal program flow. If one chooses an exception that is triggered very rarely in 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 for the backdoor is built-in by virtue of the exception-handling mechanism.

Attacking exception handling is attractive for exploitation as well as backdoors. Code surrounding exception handling and error handling in general is often not well tested. While good testing always exercises 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 frequently viewed as most important and because unlike operational states, error states are not exercised on every trial run of the software (e.g. Facebook's outage due to poor error handling [19]. Therefore, the likelihood of an attacker finding bugs (some of them constituting vulnerabilities) within these regions of a program is often increased . This fits well with an exploitation technique based on exceptions.

1.3 Contributions

We present a further step in the direction of utilizing auxiliary computations which an attacker could use to accomplish potentially malicious goals. We explain arguably the most powerful (Turing-complete by virtue of providing the same computational model as typical assembly languages), ubiquitous auxiliary environment to date: the DWARF exception handling mechanism, which comes with every modern `gcc`-compiled, exception-aware executable or shared object file. In particular, we program this automaton by way of providing it with crafted contents of the `.eh_frame` and `.gcc_except_table` ELF sections.

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 utilizing powerful bytecode

that can perform generic computations. This bytecode can, among other things, read the main process memory and register state, and in so doing make full use of the target's dynamic symbol information.

Moreover, the bytecode is quite efficient at representing such symbolic memory operations and allows us to pack much functionality into small sections of data. 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, eliminating the need to relocate or rearrange any ELF sections, a task difficult to perform on a stripped executable.

We note that the technique we present has the following properties.

1. It involves no native executable binary code, and therefore is relatively portable.

2. For the same reason, it is unlikely to be checked by any current signature-based antivirus systems (see Section 4.3).

3. It is ubiquitous in the sense that it is present wherever `gcc`-compiled C/C++ code or other exception-throwing code is supported.

4. It can bypass ASLR (Address Space Layout Randomization) through memory access and computation.

5. When 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.

We have modified Katana, our existing ELF-manipulation tool [30] to allow us to demonstrate the techniques we discuss.

Chapter 2

DWARF and Exception Handling Details

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 works as implemented by `gcc` and as partially standardized by the Linux Standards Base [1] and the x86_64 ABI [27].

2.1 Environment

All technical details discussed here are with regards to C++, `gcc`, and Linux and with specific attention paid to the x86_64 architecture. It is important to note that the applicability of the work ranges beyond this platform, however. 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. 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 [23]. All work in this study has been in the context of C++ but the same general method is used for other `gcc`-compiled languages supporting exceptions.

2.2 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, however, as it does not respect register state. Many functions contain assembly instructions before a return instruction to restore callee-saved registers appropriately before returning to the caller. This code will not run when the execution path of a procedure is interrupted by an exception. 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, since 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 so on. Therefore, the Call-Frame Information section of the DWARF standard [12] has been adopted for encoding the unwinding information necessary for exception handling — although with some minor differences, particularly in the area of pointer encoding, which are for the most part documented by existing standards [27, 1]. It should be noted, however, that the current versions of the DWARF standard at the time of this writing is version 4 and most of the DWARF 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. Indeed, although the DWARF standard provides call frame information with a version number field, in `.eh_frame` this version number is always set at "1" despite the fact that the features supported by `gcc` for `.eh_frame` have roughly kept pace with new version of the DWARF standard.

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 the 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. The mapping between DWARF register numbers and hardware registers is architecture-specific and is generally defined by the appropriate ABI. The

DWARF registers are not required to map to actual hardware registers; for example the value of 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 [12]. 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) that 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 in actual programs) 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 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, however, many empty cells and many entries duplicated down columns. Much of the DWARF call frame information standard essentially describes a compression technique, allowing DWARF data 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 (often a procedure, although there is no requirement to this effect) and describes how unwinding may be done from within that block. 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. Note that this does not mean that `.eh_frame` does not support other things introduced in DWARF 4. A diagrammatic view of CIE and FDE structure is shown in Appendix A. Each FDE contains a series of DWARF instructions of which there are two major types. Instructions of the first type each specify 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 more compact form.

2.3 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. 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 to the value computed by a DWARF expression. This consists of one or more 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 directly machine registers, although the distinction is largely irrelevant). This allows registers to be restored based on values in memory locations and on 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 as well as stack-relative addresses). 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 arbitrary arithmetic, conditional branching, and the storage of values (on a stack), 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. For an example of a DWARF expression performing computation, see the code in Listing 2.1, which finds the length of a string located just below the base of a stack frame. This code is written in a language we created called Dwarfscript, which is discussed in Section 3.2. A complete explanation of all of the instructions used can be found in the DWARF standard [12].

2.4 Exception Handlers

We have observed how the unwinding of stack frames and the accompanying register restoration is performed. 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. Every CIE includes an augmentation string, the contents of which are implementation defined. This string 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, 27]. These 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 interprets the LSDA and determines if a handler for the exception has been found.

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:
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
```

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 read 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 and non-documented. What scanty documentation exists is not codified in official sources but is only to be found on the `gcc` mailing lists in posts such as these [44, 41] or in an old Hewlett-Packard document [14] which is detailed but is either outdated or incorrect in several places,

as practical experimentation will show. The best known source of information on the format used by `gcc` is the commented 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. 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 B.

2.5 Exception Process

The code path taken during the throwing of an exception is shown in Figure 2.2. `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 the current (4.5.2) `gcc` implementation, 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.

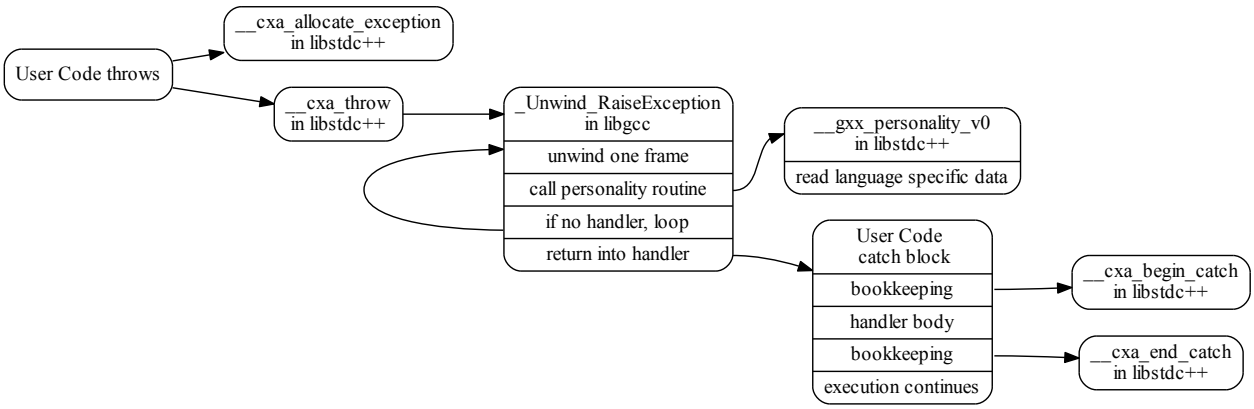


Figure 2.2: C++ Exception Code Flow

With this knowledge, the only barrier to building a payload for unex-

pected computation with DWARF is the lack of tools. We decided to address that lack.

Chapter 3

Katana and Dwarfscript

DWARF is chiefly the province of compiler and debugger authors. There are several tools that 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 that allow manipulation of DWARF structures at a higher level than a hex editor. The author of `dwarfdump` and `libdwarf` is known to be working on a DWARF generation tool which will support a text input mode, but as of this writing he has not made available any code supporting the creation of 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` is an ELF and DWARF manipulation tool that the first author wrote to aid in hotpatching research [7]. The underlying binary manipulation framework was written to be fairly flexible and was expanded to support the complex exception-handling structures used in this research 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 `Katana` can reinsert into the executable. This power allows a researcher to experiment with the consequences of carefully

crafted DWARF instructions with ease.

`Katana` is built on `libelf`[10] and `libdwarf`[3]. We chose `libelf` 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`. `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. As well as Dwarfscript, it was necessary to write the `Katana` shellscript. This turns `Katana` into a command interpreter/shell in the spirit of Elfsh [25]. 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

Full documentation of the commands available in the `Katana` shell can be found in the `Katana` documentation. `Katana`'s command set is extended frequently; current commands allow the following types of tasks:

- loading and saving ELF files and arbitrary raw data

- printing useful information about ELF files
- exporting the Dwarfscript for an ELF file with exception-handling sections (`.eh_frame`, `.gcc_except_table`)
- compiling a Dwarfscript file into ELF sections
- extracting sections and data from an ELF file
- replacing sections and data in an ELF file
- running shell commands
- performing hotpatching (not directly relevant to this research, see [7])

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 is 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 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). In all cases, care has been taken to follow the structure of the DWARF data as specified in the DWARF standard [12]. 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 a formal grammar is given in Appendix C. Familiarity with the DWARF standard should allow one to understand Dwarfscript without additional documentation, as the goal is to provide a textual representation of what would otherwise be encoded in a compact binary format. 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

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, assuming 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 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. By using DWARF expressions with logic and arithmetic, more complicated operations, such as dynamic linking, may be performed as discussed in the rest of this section.

4.1 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. The full program that we used to carry out our test is not shown here but is available upon request. Our example program simply 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 (the type of exception is irrelevant here). This program, while a toy and 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 that is interpreted as machine instructions or directly affects the linking of machine instructions. Specifically, 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`, and `.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 later in Section 4.5.1. The specific code that control 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 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 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 in this listing 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++`

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
```

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, however, 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 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 the chosen 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
```

4.2 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. 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 in which shared libraries are loaded will be the order in which they are listed in `.dynamic`. This second assumption is not crucial, since 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 [43, 21], and our DWARF implementation is not substantially different in functionality except that it is done on a stack machine rather than a register machine. 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.

4.3 Detection By Antivirus Software

It is our belief that modifications to `.eh_frame` and `.gcc_except_table` are less likely to be detected by current antivirus software than more common static manipulation of ELF binaries. As most antivirus systems are commercial and closed-source, we cannot make any definitive claims. Certain antivirus software will match against the `.text` section of known malware

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
```

if it is inserted into some other (previously harmless) program, but will not match if the same data is inserted into the `.eh_frame` section, providing circumstantial evidence for our claim. For our test we used the F-Prot and Bitdefender antivirus software available from F-Secure and Softwin respectively and Linux virus samples available from `http://vxheavens.com/`. We chose three samples detected by both F-Prot and Bitdefender: Linux.Virus.Clifax, Linux.Virus.Balrog.a, and Trojan.Linux.Attack. We wrote a simple program in C++ which performed no malicious activity and was not flagged as a virus by either scanner. Using `Katana` we created for each of our three samples a version of our program that had its `.text` section replaced with the `.text` from the virus and a version which had its `.eh_frame` replaced with the `.text` from the virus. In the case of Clifax, both AV programs flagged the program with the replaced `.text`. For Trojan.Linux.Attack, only F-Prot flagged the replaced `.text` and Bitdefender did not. For the Balrog virus, neither AV program flagged the replaced `.text` section. The replaced `.eh_frame` section was never flagged by either program for any piece of malware. This provides some circumstantial evidence that existing antivirus programs do not match their signature databases against the `.eh_frame` section. We hypothesize that detecting the sort of trojan we have demonstrated would require modification to existing antivirus software.

4.4 Combining with Traditional Exploits

What we have concretely demonstrated so far is a trojan technique. If we have a means of overwriting exception handling data or of otherwise making data we control to be interpreted as exception handling data, DWARF can be used in the construction of an exploit (rather than a trojan).

For our general technique to succeed, we need 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 interpreted when an exception is thrown. This technique aids in circumventing 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 and as a less-studied area has fewer known detection/mitigation techniques.

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` [13]. This meant that most PIC code (i.e. libraries) still required a 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. 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.

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` which is again not an easy target. `libgcc` caches program headers, however, in the variable `frame_hdr_cache`, which is static to its compilation unit. Since this is a static variable, its symbol is not exported, and 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 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`. We built a demo using a trivial format-string vulnerability (see [37, 35, 22] for background information on format-string vulnerabilities) as a proof of concept of this exploitation technique. Using the vulnerability, addresses on the stack can be leaked, and the cached location of the program header calculated. This cached value can then be overwritten using the infamous `%n` format string component [37]. This allows us to bring the trojan techniques discussed earlier into the realm of exploitation.

4.5 Limitations and Workarounds

There are several limits that anyone attempting to use the DWARF virtual machine for arbitrary computation will encounter. It is important to recognize these to plan for and work around them.

4.5.1 Registers and Parameter Passing

Not all machine registers may be restored during stack unwinding. A hardware ABI will define some set of registers which are callee-saved and some set which is 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 through DWARF to certain registers not in the callee-saved set will be ignored when restoring the call frame to return into. Some of these non-restored 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 such as 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 that 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 by innocuous means, such as expected user input) this value can be searched for by 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 construction and injection of the payload.

4.5.2 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 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 restricted contiguous stack space that can

be controlled, limited by the number of restored registers and how they are laid out in memory.

A second workaround is to take advantage of the lack of defensive programming in `libgcc` when parsing DWARF data. As can be seen in the `gcc` source file `gcc/unwind-dw2.c`, the interpretation of several DWARF instructions involves writing values read from the DWARF data to an array element indexed by another value also read from the DWARF data. No bounds checking is performed in the current implementation, and thus it is possible to provide malformed DWARF data which will achieve semi-arbitrary memory writes.

4.5.3 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 [13]. 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, this size appears to have been arbitrarily chosen as a size that should be "large enough" for any DWARF program `gcc` would expect to be necessary. 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 4.2, a dynamic linker can be programmed in DWARF using fewer than 20 words on the stack.

4.5.4 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, if overwriting an existing `.eh_frame` 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 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 from the script file, making available more room for lengthy DWARF expressions. The dynamic linker discussed earlier in Section 4.2 requires less than 200 bytes of space for its instructions.

4.5.5 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`.

4.6 Defenses

There are some fairly straightforward countermeasures to the techniques discussed in this paper. If the location of `.eh_frame` is not cached then the exploit discussed above ceases to work (assuming that `.eh_frame` is not writable). This would be a fix for a specific weakness, however, and other vulnerabilities might exist.

More fundamentally, it is not clear that the exception-handling unwind mechanism needs to have the full power that it does. `gcc` does emit register rules using DWARF expressions, but it does so infrequently and we have observed no use of branch instructions in these expressions. The `gcc` maintainers might consider either eliminating support for these instructions entirely or providing immutable flags by which a program can assert that it does not use these features. While removing Turing-complete computation would not obviate all malicious uses of exception-handling data, it would greatly reduce the power of these exploits, and thus the attractiveness of the mechanism.

Chapter 5

History and Related Work

The particular computational model that we present is to the best of our knowledge not previously discussed in a security context. There has been some prior acknowledgment of the potential misuse of exception-handling data, although it has been minimal on platforms other than Windows. The most relevant work to our research, however, is previous efforts in exploiting auxiliary computations.

5.1 Security of Exception Handling

At a high level, there is some awareness that exceptions have an impact on security. It is broadly known that by interrupting the normal flow of execution, exception handling makes a system more difficult to reason about from a security perspective because it may be infeasible to determine if all of the divergent execution paths made possible by an exception are secure [26]. Relatively recently, security concerns have been raised over the specification of exceptions for the next iteration of C++, known as C++0x. C++0x drafts specifies a `noexcept` keyword which allows the programmer to indicate that a function or expression will not throw an exception [18]. Unfortunately, in certain versions of the draft (it is of course impossible at present to know with certainty what the final standard will hold) the behavior if an exception *is* thrown within a function marked as `noexcept` is undefined. Kohlbrenner et al. demonstrate how this behavior could be exploited in one possible implementation for Visual C++ [20]. While it appears likely that the final standard may require a call to `std::terminate` in such circumstances,

eliminating undefined behavior, the issue still highlights the sorts of flaws in exception handling, whether standards-compliant or not, that could potentially be exploited.

On Linux/Unix systems, exploiting exception handling is mostly an unstudied topic. The implementation of exception handling on Windows is much different, where C++ code compiled by Visual Studio uses the Windows-specific exception handling mechanism known as Structured Exception Handling (SEH) [28]. There is some support for SEH within gcc on Windows as well. SEH involves the storage of exception handler information on the stack, as part of stack frames. This makes it a relatively easy target, and attacks on SEH are readily found in academic literature [5] and the online hacker community [16]. These attacks prompted Microsoft to introduce a new compiler flag enabling stack-based exception protection [15].

As `gcc` on Linux does not use any extra stack information for exception handling, exception handling on Linux is a more difficult target and has not been thoroughly examined. It has been hypothesized in passing by Younan et al. that overwriting `.eh_frame` could allow an attacker to redirect execution to injected or chosen code [46], but to the best of our knowledge this hypothetical attack has not been demonstrated and they did not recognize that computation could be carried out by the exception handling mechanism itself.

5.2 Historical Exploitation

The original focus of mainstream exploitation was on inserting native binary code directly [2]. With the advent of W⊕X protections as pioneered by the PaX and Openwall projects [33, 32], attackers' attention turned to manipulating data that controlled code and function addresses rather than inserting or modifying code directly. While preventing the insertion of arbitrary machine code executable by the processor is relatively straightforward, especially with hardware support, detecting and preventing the case in which a program is executed with unexpected data is considerably less straightforward, and indeed is believed by the PaX team to be an intractable problem [34]. Return-to-libc exploits through buffer overflows [40] are perhaps the first major example of exploitation through using data to control the flow of execution. Code originally in the process image is "borrowed" to be called out of order by the attacker.

The advent of format-string exploitation introduced the idea of "programming" with a "language" other than machine code in exploit development [37, 35]. Format string primitives comprise a simple language with which one can read and write data anywhere within the target. This idea is key. There is no fundamental difference between code and data when the computation carried out by the "code" is driven by inputs (the "data"). Machine code is no more and no less than input data for the hardware CPU. Nearly every piece of data is the input to some automaton; it is "instructions" programming that automaton. Format strings are interpreted by an automaton (albeit not Turing-complete) and if the format-string automaton can be made to interact arbitrarily with writable data in a process address space, then an attacker is able to program the main computation even if he may not touch the machine code. Heap-block management logic is another such automaton that can be co-opted [4].

The more recent significant development in data-driven exploitation is return-oriented programming [38, 9, 17]. ROP presents the opportunity to build a Turing-complete "gadget" set using borrowed code fragments chained together by return or jump instructions through control of the stack. The notion of a gadget set brings home the realization that data is not fundamentally different than code and that exploitation is no more nor less than the programming of unexpected machines. Indeed, work by Dullien et al. shows that gadget sets can be found automatically and that ROP exploits can be written in a gadget-independent language which is compiled into the discovered gadget set [11].

5.3 Auxiliary Computation

As exploits based on modifying the stack or heap used by the main computation developed, side-by-side exploits emerged using auxiliary computation systems present in the ELF linker and loader. Some of these are quite simple: i.e. overwriting lists of functions to be called during process setup/teardown in `.ctors` and `.dtors` [36, 24]. More deviously, the PLT (Procedure Linkage Table) and GOT (Global Offset Table) tables used for dynamic symbol resolution by the dynamic linker can be overwritten to point functions to chosen locations much as any function pointer used in the main computation can be so overwritten [8, 31, 45]. Perhaps most significant is the realization that it is not necessary for an attacker to do all the work of bypassing

ASLR and other modern protections if well-defined mechanisms exist for accomplishing the task. As shown by Nergal and others, the dynamic linker can be co-opted by an attacker for symbol resolution without resorting to more difficult forms of information leakage to defeat ASLR [29, 42]. While a proof-of-concept code obfuscation tool and not an exploitation mechanism, Locreate [39] demonstrates the power of the standard relocation system to modify the text (or any other section) of a program at load time. Through the use of these auxiliary systems, attackers can often gain information about and control over a process more easily than is always possible by focusing on the main computation alone, and we argue that the exception handling mechanism is an auxiliary system which should not be ignored.

5.4 Where Does DWARF Fit In?

ROP is the realization of Turing-complete "data" languages interpreted by the main computation. While the finding of these languages can be automated, this work is unnecessary if a well-defined Turing-complete language already exists. Therefore, we present DWARF overwriting as a potential alternative to ROP that an attacker may take advantage of in code that makes use of exceptions. While it can be more challenging to set up than ROP, as the cached program header to overwrite discussed earlier in Section 4.4 will generally require first leaking the `libgcc` load address, exploits can be written in an existing Turing-complete environment, eliminating the need to built a gadget set. Additionally, defenses against ROP are beginning to arise [6], and as a new technique this may remain undefended against for longer.

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 for attackers over traditional backdoors and over return-oriented-programming. Notable features 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, except of course for a data overwrite vulnerability used to inject crafted DWARF data.

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. 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.

6.1 Availability

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

6.2 Acknowledgments

Significant portions of the text of this document are based on work submitted by James Oakley and Sergey Bratus to USENIX WOOT '11. Notification of acceptance (or lack thereof) to WOOT is still pending. Much of the material is also based on presentations made by James Oakley and Sergey Bratus earlier in 2011 at the security conferences Shmoocon, Hackito Ergo Sum, and PH-Neutral. The author would like to thank the organizers of those conferences for the environment of security research ideas they have fostered, with thanks especially to Felix "FX" Lindner for his vision and support.

I owe a debt of thanks to the people who have helped me along the way. These include, but are not limited to, the following: my advisor Professor Sergey Bratus, without whom the project would not have been possible; Rodrigo Rubira Branco and Travis Goodspeed for advice on modern exploitation techniques; Professor Sean Smith for support and advice; the entire Dartmouth College Trust Lab for being full of wonderful and intelligent people; and Tim Yardley for useful discussions. Thanks also to those who helped with the editing of this document including Amber Gode, Professor Sean Smith, and Anna Shubina.

This material is based in part upon work supported by the Department of Energy under Award Number DE-OE0000097. This report was prepared as

This material is based in part 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.

Bibliography

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

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine 7*, 49 (1996).

[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] Anwar, P. Buffer overflows in the microsoft windows environment. Tech. rep., Royal Halloway University of London, February 2009.

[6] Bania, P. Security mitigations for return-oriented programming attacks. Whitepaper, Kryptos Logic Research, 2010.

[7] 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.

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

[9] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.

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

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

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

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

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

[15] Howard, M. Protecting your code with visual c++ defenses. *MSDN Magazine*, March (2008).

[16] Hubener, D. Understanding SEH (structured exception handler) exploitation. http://www.i-hacked.com/content/view/280/42/, July 2009.

[17] 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.

[18] ISO/IEC. *ISO/IEC FCD 14882, Programming Language C++*, n 4512 ed., 03 2010. (C++0x Draft Standard).

[19] Johnson, R. More details on today's outage. http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919, Sep. 2010. This document is a post from a Facebook employee describing an outage.

[20] Kohlbrenner, D., Svoboda, D., and Wesie, A. Security impact of noexcept. ISO/IEC C++ Programming Language – Core Working Group.

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

[22] Lhee, K.-S., and Chapin, S. J. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience 33*, 5 (2003), 423–460.

[23] 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.

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

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

[26] McGraw, G. From the ground up: The dimacs software security workshop. *IEEE Security and Privacy 1*, 2 (2003), 59–66.

[27] 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.

[28] Microsoft Corporation. Structured exception handling (C++). *MSDN* (2010).

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

[30] Oakley, J., and Bratus, S. Katana hotpatching tool. http://katana.nongnu.org, June 2010.

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

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

[33] PaX Team. PaX project. http://pax.grsecurity.net/.

[34] PaX Team. PaX future. http://pax.grsecurity.net/docs/pax-future.txt, 2003.

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

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

[37] scut, and team teso. Exploiting format string vulnerabilities, 2001. No canonnical source; widely available on the Internet.

[38] 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.

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

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

[41] 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.

[42] the grugq. Cheating the ELF: subversive dynamic linking to libraries.

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

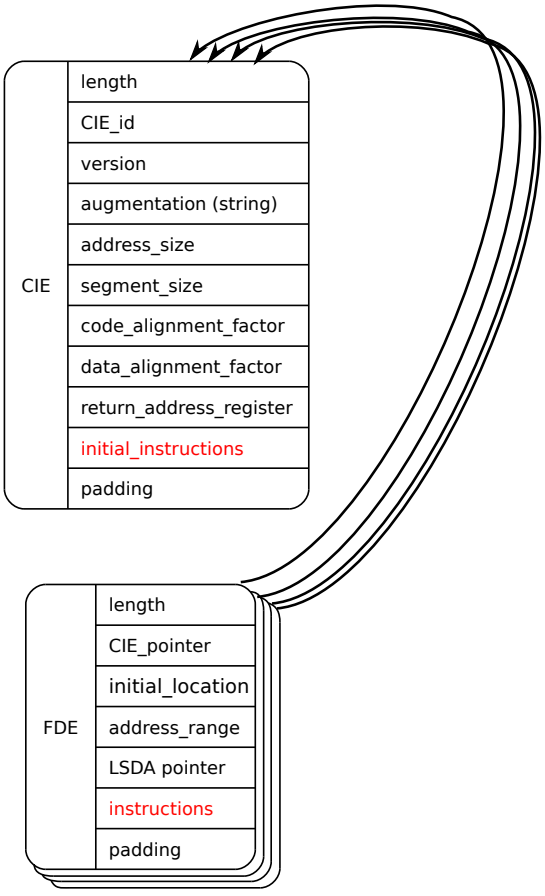
[44] Various. gcc@gcc.org mailing list: Exception handling description. http://gcc.gnu.org/ml/gcc/2009-05/msg00390.html, May 2009.

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

[46] Younan, Y., Joosen, W., and Piessens, F. A methodology for designing countermeasures against current and future code injection attacks. *Innovative Architecture for Future Generation High-Performance Processors and Systems, International Workshop on 0* (2005), 3–20.

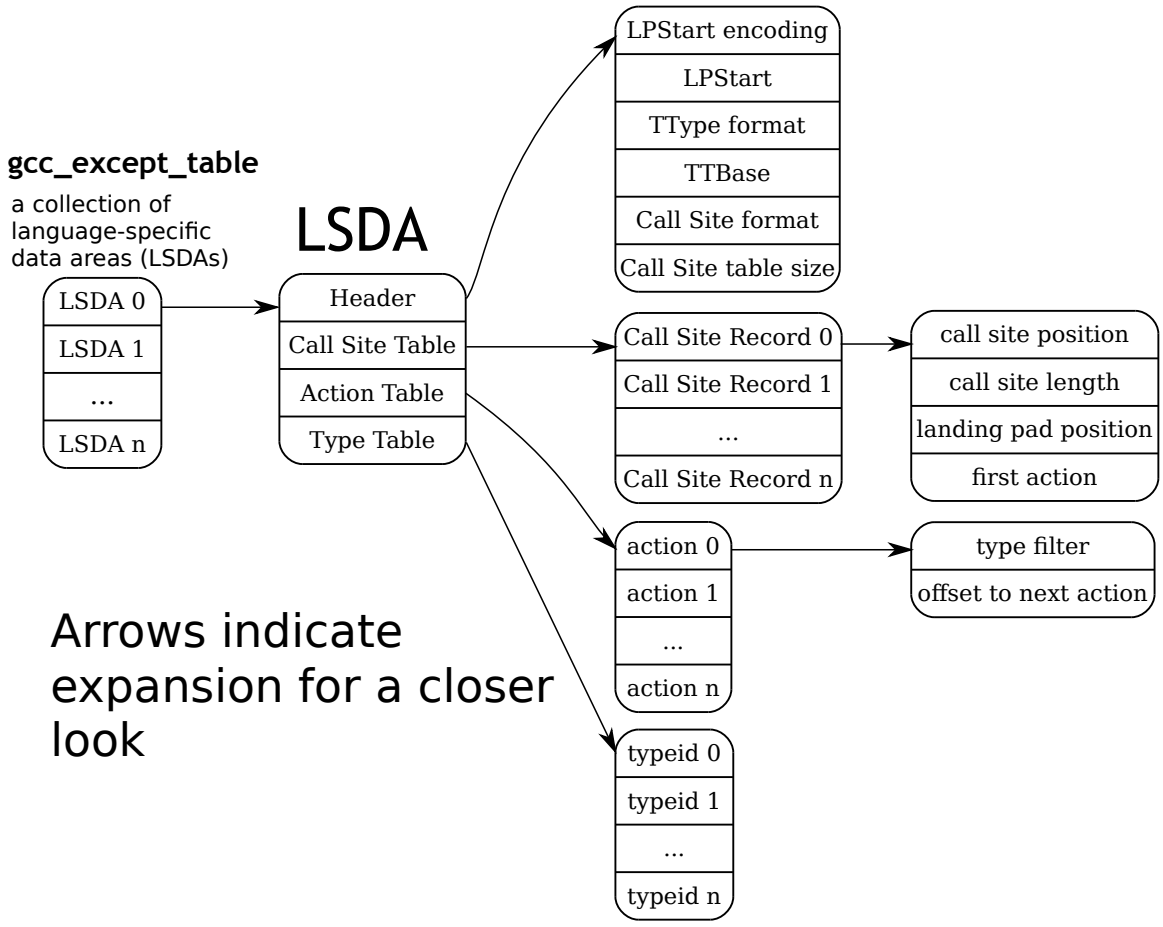
Appendix A

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>
|<eh_hdr_table_enc_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>

<eh_hdr_table_enc_prop> → **eh_hdr_table_enc** **:** <dw_pe_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>
| **type_idx** **:** **match_all**

<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>