Overwriting the Exception Handling
Cache Pointer - Dwarf Oriented Programming

James Oakley (Electron)
Electron100 *noSPAM* gmail.com

Rodrigo Rubira Branco (@BSDaemon)
rodrigo *noSPAM* kernelhacking.com

Sergey Bratus (Sbratus)
Sergey *noSPAM* cs.dartmouth.edu
Credits

• This presentation combines ideas, research, discussions from the following personnel:
  – Sergey Bratus (Insecurity Theory, Exploiting the Hard-working Dwarf)
  – Meredith Patterson (Langsec)
  – R.I.P. Len Sassaman (Langsec)
  – James Oakley (Exploiting the Hard-working Dwarf -> everything related to that, including Katana)
  – Rodrigo Rubira Branco (Exploiting the Hard-working Dwarf -> exploitation, implementation details, research organization)
Motivation

- Software exploitation is not generic anymore
- There are different exploitation primitives in different contexts
- A modern exploitation technique shows how to take advantage of those primitives
- There are much more been ‘computed’ then only the code that was written (the target)
Insecurity State

• “Treat all systems as compromised”
  – “There's no such thing as ‘secure’ any more.” -- Deborah Plunkett, NSA Information Assurance Directorate

• “Long weeks to short months before a security meltdown” – Brian Snow, in December 2010
Does prevention works?

• Many initiatives in ‘trustworthy computing’
• Many books on ‘secure programming’
• Many academic publications
• Lots of test methods: fuzzing, binary analysis, you name it
  – And STILL the software SUCKS and is EXPLOITED!
• And about the hardware? We have no idea how bad it is...
Insecurity is all about computation

• *Trustworthiness* of a computer system is what the system can and cannot compute
  – Can the system decide if an input is invalid/unexpected/malicious & reject it?
  – Will program perform only *expected* computations, or *malicious* ones too?

• *Exploitation is setting up, instantiating, and programming a weird machine*
  – A part of the target is overwhelmed by crafted input and enters an *unexpected* but *manipulable* state
Decidability

• Computation has some unsolvable (un-decidable) problems – about recognition of inputs!

• An un-decidable problem does not have an algorithm that solves it for the general case
Software Exploitation

• A part of the target is overwhelmed by crafted input and enters an unexpected but manipulable state.

• **Primitives** are exposed
  – Memory corruption, implicit control flows
  – Unexpected control flows, ...

• A “weird machine” is unleashed
  – A more powerful, programmable execution environment than intended or expected
Software is Complex

• Checks for input validity are scattered throughout the program, mixed with processing logic

• Lots of additional computing options existent and available to the ‘weird machine programmer’, aka, exploit writer
Weird Machine is Born!
Exploiting Additional Computations

• Finally we are in our talk line...

• There are many computations inside a program that can be used to subvert the code execution (and some of them has nothing to do with the original code itself)

• ROP is not new, exploits are using it since 2000 (maybe even before)
*nix Exception Handling

• Binaries compiled with GCC and that support exception handling have Dwarf bytecode:
  – Describe the stack frame layout
  – Interpreted to unwind the stack after an exception occurs

• The process image includes the Dwarf interpreter (part of the GNU C++ runtime)

• Bytecode can be written to force the interpreter to perform any computation (Turing-Complete), including, but not limited to, setup a library/system call modifying registers such as stack and base pointers → See James and Sergey previous work on Dwarf Trojans
James Oakley and Sergey Bratus

• Proved that Dwarf can replace code creating a Trojan completely using Dwarf bytecode

• Proved that Dwarf is a complete development environment:
  – Can read memory
  – Can compute with values from memory/registers
  – Can influence the flow of execution of a process
The executable has this format either on disk or in memory.
Dwarf

- Developed as a debugging format to replace STABS

- Standard:  [http://dwarfstd.org](http://dwarfstd.org)

- Provide information such as code line, variable types, backtraces, others

- ELF Sections:  `.debug_info`, `.debug_line`, `.debug_frame` are defined in the standard

- `.debug_frame` defines how to unwind the stack (how to restore each entry in the previous call frame)
Linux Exception Handling

• GCC, the Linux Standards Base and the ABI x86_64 adopted a very similar format used in the .debug_frame to describe the stack unwind during an exception: .eh_frame

• It is not exactly the same as dwarf

• It adds pointer encoding and language-specific data

• As usual, the documentation is sparse and very limited:
  – Partially discussed in the Linux Standards Base
  – Partially defined in the ABI
  – Partially implemented in GCC
Theoretically it is a table, where for each address in the .text it is describe how to restore the registers to the previous call frame

<table>
<thead>
<tr>
<th>EIP</th>
<th>CFA</th>
<th>EBP</th>
<th>EBX</th>
<th>EAX</th>
<th>RET</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>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>

CFA (Canonical Frame Address) – Address relative to the call frame

Each line defines how each part of the code can return to the previous frame
Size Limitations

• Obviously, keep such a table would use more space then the code itself
• That’s why the adoption of bytecode: The table is ‘compressed’, providing everything required to create it when needed
• Portions of the table are created as needed (on-demand)
"The .eh_frame section shall contain 1 or more Call Frame Information (CFI) records. The number of records present shall be determined by size of the section as contained in the section header. Each CFI record contains a Common Information Entry (CIE) record followed by 1 or more Frame Description Entry (FDE) records. Both CIEs and FDEs shall be aligned to an addressing unit sized boundary"
FDE x CIE

FDE (Frame Description Entry) exists for each logical Instruction block

CIE (Common Information Entry) holds common Information between FDEs

INSTRUCTIONS in FDE hold the DWARF bytecode
FDE x CIE

**initial_location/address_range:**
Defines for which instructions this FDE applies

**augmentation:**
Language-specific information

**return_address_register:**
Entry in a virtual table that defines the .text location to return to (eip)

**instructions:**
Table rules. Dwarf has a language to describe the table.
Dwarf Instructions

• Work as an assembly language (unexpected computations)

• Turing-Complete Stack-Based Machine

• Can access memory and register values

• Have some limitations:
  – Cannot write to register/memory (but we can force out-of-order code execution and obtain writes)
  – Cannot call native code
  – Cannot write to registers that are not callee-saved in the ABI (we can write to callee-saved register thought)
  – GCC limits the stack in 64 words
Dwarf Programming

- **DW_CFA_set_loc N**
  Next instructions apply to the first N bytes of the function

- **DW_CFA_def_cfa R OFF**
  CFA is calculated starting from register R and offset OFF

- **DW_CFA_offset R OFF**
  Register R is restored from the value in CFA OFF

- **DW_CFA_register R1 R2**
  Register R1 is restored with the contents of R2
And the table is back...

• Each architecture register receives a DWARF equivalent (the mapping is architecture specific)
• Dwarf Instructions define rules for a column or advances to the next line (program location)
• In a FDE, lines heritage from instruction lines above them

<table>
<thead>
<tr>
<th>EIP</th>
<th>CFA</th>
<th>EBP</th>
<th>EBX</th>
<th>EAX</th>
<th>RET</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>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

- To not anticipate all unwinding mechanisms of a system, the standard defines flexibility:
  - DW_CFA_expression R EXPRESSION
    R receives the value from the EXPRESSION result
  - DW_CFA_val_expression R EXPRESSION
    R restored to result of EXPRESSION

- Expressions have their own instructions:
  - Constant Values: DW_OP_constu, DW_OP_const8s, etc
  - Arithmetic: DW OP plus, DW OP mul, DW_OP_and, DW_OP_xor, etc
  - Memory read: DW_OP_deref
  - Register read: DW_OP_bregx
  - Flow Control: DW_OP_le, DW_OP_skip, DW_OP_bra, etc
Katana

**Emit a dwarfscript**
- $e=load "demo"
  
  Loaded ELF "demo"

- dwarfscript emit ".eh_frame" $e "demo.dws"
  
  Wrote dwarfscript to demo.dws

**Dwarfscript assembler**
- $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
So what?

- With Katana you can see and modify unwind tables in an easy way
  - Control the unwinding flow (how the call stack is handled)
  - Avoid an exception handler to execute another one
  - Redirect exceptions
  - Find/solve symbols
  - Calculate relocations
Example

• If function foo is responsible for an exception
  – Change flow to function bar
  – Thru static analysis, we see that bar is at 0x600DF00D
  – In the FDE, we change:
    DW_CFA_offset r16 1
  – To:
    DW_CFA_val_expression r16
    begin EXPRESSION
    DW_OP_constu 0x600DF00D
    dnd EXPRESSION
    EXPRESSION
.gcc_except_table

• So far, redirected only to ‘catch’ blocks

• The .gcc_except_table hold language-specific data (where the exception handlers are)
  – Interpreted by the personality routines
  – We can stop an exception at any time
  – Unlike the .eh_frame, do not have standards
  – There is no documentation, so let’s see the code ;)
Assembly

- While compiling a program using GCC, do:
  --save-temps –fverbose-asm –dA

  .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 _ZTli
Layout

gcc_except_table
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
- ... (n)

action 0
- action 1
- ... (n)

typeid 0
- typeid 1
- ... (n)

call site position
- call site length
- landing pad position
- first action
- type filter
- offset to next action
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

__cxa_begin_catch in libstdc++

__cxa_end_catch in libstdc++
Exceptions are not asynchronous

• Functions that call throw() just call:
  – __cxa_allocate_exception() -> To allocate space using malloc (or buffers in the .bss if malloc fails – gcc-xxx/libstd++v3/libsupc++/eh_alloc.:84)
  – And then __cxa_throw() -> That will go thru the frames until a handler for the exception is found
Proving (assembly)

Dump of assembler code for function main:
...
>+9>: mov $0x4,%edi       # std::size_t thrown_size

  # Allocates a new "__cxa_refcounted_exception" followed by 4 bytes; we
  # do a "throw(1)", 1 being an "int" occupies 4 bytes.
>+14>: callq 0x400930 <__cxa_allocate_exception@plt>
...
>+25>: mov $0x0,%edx       # void (*dest) (void *)
>+30>: mov $0x6013c0,%esi    # std::type_info *tinfo
>+35>: mov %rax,%rdi        # void *obj
>+38>: callq 0x400940 <__cxa_throw@plt>
__cxa_allocate_exception()  

• Returns a pointer to a  
  – struct __cxa_refcounted_exception, which holds a reference to an object __cxa_exception  

• __cxa_throw() is then executed to:  
  – Initialize the current context (register values)  
  – Iterate in the stack until it finds the exception handler
What We've Shown Before

- Ret-into-libc
- Used the dynamic-linker already in Dwarf to find execvpe
- Used Dwarf to prepare the stack
- In less than 200 bytes and less than 20 words in the stack (showing that a 64-stack word limitation is not an obstacle)
- Started in an offset of execvpe where they can control the Dwarf registers (and not in the function beginning)
What else can be done?

• Old GCC had both, the .eh_frame and the .gcc_except_table as +W

• Well…
  – Libgcc/libstdc++ need to find those areas in memory, right?
  – The program header, GNU_EH_FRAME contains the .eh_frame location (dl_iterate_phdr is the function that finds it)
  – Libgcc caches the value!
Fake EH

• If we can overwrite the cached value, we are able to control the exceptions and leverage everything already explained

• Libgcc does not export symbols, so we need to find an heuristic/reverse to find what to overwrite
Caching

- The pointer caching is done in: unwind-dw2-fde-glibc.c:
  
  ```c
  #define FRAME_HDR_CACHE_SIZE 8
  ...
  static struct frame_hdr_cache_element
  {
    _Unwind_Ptr pc_low;
    _Unwind_Ptr pc_high;
    _Unwind_Ptr load_base;
    const ElfW(Phdr) *p_eh_frame_hdr;
    const ElfW(Phdr) *p_dynamic;
    struct frame_hdr_cache_element *link;
  } frame_hdr_cache[FRAME_HDR_CACHE_SIZE];
  ```
Caching

• 8 cache entries for the frame header
  – Uses a Least Used Replacement Algorithm (_Unwind_IteratePhdr_Callback())
  – Most recently used is the head of the list

• In the test environment, the frame_hdr_cache was at 0x6e0 bytes from the offset of the writable data segment of libgcc

• This is the aforementioned array, with 48 bytes in size

• The executable itself is the 3rd element of the array (the first two are the libgcc and libstdc++)

• The offset for the writable data segment of libgcc can be found in this way (based in what we know):
  – 0x6e0+48*2=0x740

• The entry p_eh_frame_hdr that we want to overwrite is at 24 bytes of this structure.
Example

• Together with the paper/presentation, we release a demo program for the exploitation. Those are the test characteristics we see:
  – 0x7ffff760e000 -> libgcc loading address
  – 0x220000 -> offset for the writable data segment (starting from the lib base address)
  – 0x6e0 -> offset for the cache elements (starting from the writable data segment)
  – 48 bytes is the size of the structure
  – 2 entries before ours (readelf –d)
  – 24 bytes inside the structure is what we want to overwrite
Exploiting

• To simplify the exploitation, it is interesting to align the structures in known offsets/controlled offsets:
  – .eh_frame in the example aligned to start exactly at 0x50 bytes from the start of the .eh_frame_hdr
  – .gcc_except_table aligned to start exactly at 0x200 bytes from the start of the .eh_frame
Memleak

- We need the value of EBP, so we going to use a memleak. It can be achieved in different ways, depending on the target program (e.g.: overwriting parameters to printf-like functions, or if the vulnerability is a format string, which is our sample case)

- To calculate the EBP_PREVIOUS we use %llx (format string), so we use 4 bytes of space in the buffer and advance the stack pointer in 8 bytes (so the premise for exploiting the sample program is to manage to leak the EBP):

```python
...  #to get the value of ebp_previous
instr=r"%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%llx%x%x"
proc.sendline(instr)
proc.expect("unknown command: [0-9a-f]* ([0-9a-f]*).*")
ebp_previous=int(proc.match.group(1),16)
info("\nfound ebp_previous = 0x%x" % ebp_previous)
```
Heuristics

• We know the size of the previous frame (disassembling), so we are capable of calculating the EBP of our frame:
  – ebp=ebp_previous-PREV_FRAME_SIZE

• With our frame address, we can calculate the address of libgcc, since we know the offsets:
  – libgcc_reveal_location=ebp-LIBGCC_REVEAL_EBP_OFFSET;
More Heuristics

- The value that reveals the .text location of the libgcc is at 0xffffffc798 (discovered in the previous slide), and it is 0x679 above ESP and 0x750 above EBP

- The libgcc base is calculated using the previously revealed address and masking the 3 low nibbles. We also use a fixed value to adjust the result (found thru disassembly):
  - \( \text{libgcc\_base} = (\text{libgcc\_revealed} \& 0xFFFFFFFF) - \text{LIBGCC\_REVEAL\_ADJUST} \)

- The separation between .text and .data segments in libgcc is 0x19000 (x86):
  - \( \text{libgcc\_data\_base} = \text{libgcc\_base} + \text{LIBGCC\_DATA\_OFFSET} \)
Finalizing

• Finally, we find the frame_hdr_cache and the respective p_eh_frame_hdr from the libgcc_data_base, as previously described:
  – frame_hdr_cache=libgcc_data_base+CACHE_LIBGCC_OFFSET
  – p_eh_frame_hdr=frame_hdr_cache+CACHE_ENTRY_SIZE*PREVIOUS_CACHE_ENTRIES
    +OFFSET_IN_CACHE_ENTRY
In the demo case

• With all the values, we redirect the execution:
  – Function doWork starts at 0x0804936a
  – Throw is at 0x08049634
  – Distance: 0x2ca (call site 14 in the dict_mod.dws for the demo)
  – We force the execution of the I_am_never_called that is at 0x08049842 (offset of 0x4d8 from doWork)

• The Dwarf payload is injected in the dictionary been readed by the target program (instead of using a shellcode). We find the pointers, overwrite the caching target address and the desired catch block is executed.
Other possibilities

• If you have a Write N you can overwrite the .eh_frame entirely (if it is +W, what is not normal in new systems)

• You can overwrite the .eh_frame using a shellcode

• You can use a staged ret-into-lib to remap the .eh_frame as +W and then overwrite it
THE END! Really is !?

James Oakley (Electron)
Electron100 *noSPAM* gmail.com

Rodrigo Rubira Branco (@BSDaemon)
rodrigo *noSPAM* kernelhacking.com

Sergey Bratus (Sbratus)
Sergey *noSPAM* cs.dartmouth.edu