Vulnerability Analysis I:
Exploit Hardening Made Easy
Surgically Returning to Randomized Lib(c)

Mitchell Adair

September 9th, 2011
Outline

1. Background
2. Surgically Returning to Randomized lib(c)
3. Exploit Hardening Made Easy
1. Background

2. Surgically Returning to Randomized lib(c)

3. Exploit Hardening Made Easy
A Basic Exploit

**Taking** over control flow

1. Attacker **writes** data into memory
2. Attacker overwrites control structures
3. New values in control structures point to attackers malicious payload
4. Program execution eventually transfers and **executes** malicious payload
A Basic Exploit

ESP

NOP sled

Payload (/bin/sh)

EBP

ret

arguments...

EBP

local variables

RET

controll structures
Data Execution Prevention

**DEP**

Marks regions of memory as writeable or executable, not both

- Prevents attacker from writing a payload to memory, then directly executing it
Data Execution Prevention

ESP

EBP

ret

local variables

controll structures

arguments

...
What Now?!

Are we defeated because of DEP?

- No, a clever technique called Return Oriented Programming (ROP) emerged to combat DEP
- ret2libc - returning to lib(c) is a special case of return oriented programming
Return Oriented Programming

ROP Leverages

1. The fact that the "ret" instruction will pass control flow to an arbitrary memory address pointed to by ESP
2. Small snippets of code that end in a "ret" instruction - called gadgets.

Gadgets

- Gadgets can be built up consecutively on the stack, executing small groups of instructions before returning control flow to the next gadget
- Gadgets are found in executable sections of libraries or the program itself, therefore avoiding execution directly on the stack
Return Oriented Programming

Credit: [Exploit Hardening Made Easy, USENIX Security 2011]
Returning to libc

A special case of ROP

- The saved return value on the stack is overwritten with the address of a function from the libc library
  - system, execv, etc.
- Items on the stack directly below the libc pointer are the arguments for the target function
Returning to libc

Credit: [Surgically Returning to Randomized lib(c)]
Address Space Layout Randomization

ASLR

Areas of memory are now located at "randomized" locations

- Makes it difficult for the attacker to know what values to overwrite control structures with
- A gadget or library call will be at one location during one execution of the program, and another location the following execution

The Caveat

In Linux, everything **but** the application itself has it's address randomized
Outline

1. Background

2. Surgically Returning to Randomized lib(c)

3. Exploit Hardening Made Easy
Goal

Be able to successfully determine and hit a target library function in one try, no brute forcing, on a system with DEP and ASLR enabled

Solution

- Use an information leak to gather data about a randomized base address of a library
  - Available directly in the program address space, unavoidable
  - Single shot "surgical precision"
  - Works on 32 bit and 64 bit systems alike
Areas of Interest

The executable contains two special data structures (or sections) used specifically for the purpose of linking the executable with shared objects:

**Procedure Linkage Table (PLT)**
- An array of jump stubs
- The $i^{th}$ PLT entry contains a jump instruction that jumps to the address stored in the $i$th GOT entry.

**Global Offset Table (GOT)**
- The linker fills the GOT with the addresses of the imported functions, updated to be consistent with the current base address of the library.
Areas of Interest

Credit: [Surgically Returning to Randomized lib(c)]
The Attack

Overall Approach

- Knowledge of the absolute address of a single function in libc is enough to mount a successful attack
  - the ability to call any function in the library
- Exploit information found in the GOT to:
  1. calculate the base address of libc
  2. calculate the absolute address of an arbitrary function
  3. invoke the function
The Attack

Definitions

**offset(s)** is a function that computes the virtual offset of symbols, relative to the base address of the library

- The virtual offset can be computed off-line from the library file and that the offset is constant.

**open** is any function used to de-randomize the library

**system** is a function whose absolute address the attacker wants

Calculations

```
libc = open - offset(open)
system = open - offset(open) + offset(system)
```
The Attack

What now?

- We now can calculate the base address of libc, and the absolute address of system
- We use ROP, using gadgets within the application itself to take advantage of these values
Attack #1 - GOT Dereferencing

Overview

1. calculate the offset between open and system.
2. Add the difference and the absolute address of open to a location.
3. Call the location.
Attack #1 - GOT Dereferencing

Setting up the registers

\[ \text{eax} = \text{offset(system)} - \text{offset(open)} \]
\[ \text{ebx} = \text{got(open)} \]

Definition

\text{got(open)} is the address of the GOT entry of open
Attack #1 GOT Dereferencing

Credit: [Surgically Returning to Randomized lib(c)]
Overview

1. calculate the offset between open and system
2. add the difference to the GOT entry of open
   (overwritting the GOT entry)
3. call open

Note

The separation between PLT and GOT is for improved security. PLT is executable, GOT is writeable.
Attack #2 GOT Overwritting

Credit: [Surgically Returning to Randomized lib(c)]
How do we prevent this?
Prevention

Position Independent Executable

**PIE** executables can be loaded at arbitrary memory locations, similar to ASLR. PIE prevents all of the previously mentioned attacks, because no ROP gadgets can be gathered from the executable itself.
But...

PIE is not used in industry. PIE requires recompilation, it incurs a small overhead, and vendors are not aware of its real importance.
Prevention

The author’s (temporary) solution

- Encrypt the GOT so only the linker and PLT can decrypt it, and make it read only. This does **not** require recompilation, but does incur a small overhead as well.
- This is stated as a temporary solution until vendors embrace PIE, however.
Prevention

<table>
<thead>
<tr>
<th></th>
<th>GOT dereferencing</th>
<th>GOT overwriting</th>
<th>Requires recompilation</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>W⊕X and ASLR</em></td>
<td>–</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td><em>Periodic re-randomization</em></td>
<td>–</td>
<td>–</td>
<td>Yes</td>
</tr>
<tr>
<td><em>GOT randomization</em></td>
<td>–</td>
<td>–</td>
<td>No</td>
</tr>
<tr>
<td><em>GOT read-only</em></td>
<td>–</td>
<td>✓</td>
<td>No</td>
</tr>
<tr>
<td><em>PIE</em> [7]</td>
<td>✓</td>
<td>✓</td>
<td>Yes</td>
</tr>
<tr>
<td><em>Self-randomization</em> [15]</td>
<td>✓</td>
<td>✓</td>
<td>Yes</td>
</tr>
<tr>
<td><em>Encrypted GOT</em></td>
<td>✓</td>
<td>✓</td>
<td>No</td>
</tr>
</tbody>
</table>

Credit: [Surgically Returning to Randomized lib(c)]

- PIE and the authors solution both stop the previously mentioned attacks, but they both still have an overhead...
There is actually very little overhead on average, only applications that call library functions with a very high frequency have a significant overhead.
Conclusion

- Leveraging information leaks from the PLT and GOT an exploit can be developed to bypass DEP and ASLR
  - Results in a single shot attack, no brute forcing
- Position dependent executables basically nullify existing security measures such as DEP and ASLR, yet PIE and other options are not yet embraced by industry
- A new solution, encrypting the GOT is proposed by the authors
  - Does not require recompilation
1  Background

2  Surgically Returning to Randomized lib(c)

3  Exploit Hardening Made Easy
Exploit Hardening Made Easy

About

- USENIX Security 2011
- Carnegie Mellon
Overview

Introduce Q
- A system for automatic exploit hardening
  - Automatically create ROP payloads
  - Automatically transform unharded exploit to hardened

Definition
**hardened** means an exploit can bypass DEP and ASLR
Phase 1

Gadget Discovery

- Use semantic program verification
  - analyze what a gadget does given random values, and classify it
- List of found gadgets creates it’s own Instruction Architecture Set (IAS), where each gadget represents an instruction
### Phase 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Parameters</th>
<th>Semantic Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPG</td>
<td>—</td>
<td>—</td>
<td>Does not change memory or registers</td>
</tr>
<tr>
<td>JUMP G</td>
<td>AddrReg</td>
<td>Offset</td>
<td>EIP ← AddrReg + Offset</td>
</tr>
<tr>
<td>MOVE REG G</td>
<td>InReg, OutReg</td>
<td>—</td>
<td>OutReg ← InReg</td>
</tr>
<tr>
<td>LOAD CONST G</td>
<td>OutReg, Value</td>
<td>—</td>
<td>OutReg ← Value</td>
</tr>
<tr>
<td>ARITHMETIC G</td>
<td>InReg1, InReg2, OutReg</td>
<td>◊&lt;sub&gt;b&lt;/sub&gt;</td>
<td>OutReg ← InReg1 ◊&lt;sub&gt;b&lt;/sub&gt; InReg2</td>
</tr>
<tr>
<td>LOAD MEM G</td>
<td>AddrReg, OutReg</td>
<td># Bytes, Offset</td>
<td>OutReg ← M[AddrReg + Offset]</td>
</tr>
<tr>
<td>STORE MEM G</td>
<td>AddrReg, InReg</td>
<td># Bytes, Offset</td>
<td>M[AddrReg + Offset] ← InReg</td>
</tr>
<tr>
<td>ARITHMETIC LOAD G</td>
<td>OutReg, AddrReg</td>
<td># Bytes, Offset, ◊&lt;sub&gt;b&lt;/sub&gt;</td>
<td>OutReg ◊&lt;sub&gt;b&lt;/sub&gt; ← M[AddrReg + Offset]</td>
</tr>
<tr>
<td>ARITHMETIC STORE G</td>
<td>InReg, AddrReg</td>
<td># Bytes, Offset, ◊&lt;sub&gt;b&lt;/sub&gt;</td>
<td>M[AddrReg + Offset] ◊&lt;sub&gt;b&lt;/sub&gt; ← InReg</td>
</tr>
</tbody>
</table>

Credit: [Exploit Hardening Made Easy, USENIX Security 2011]

- Classification of gadgets supported by Q
Phase 2

Gadget Arrangement

- Piece together gadgets to accomplish a target goal
  - such as a memory write, or read
- Wide classification of gadgets allows for a very flexible arrangement of gadgets
Phase 3

Gadget Assignment

Make sure the arranged gadgets are using corresponding registers / addresses, and at the same time don’t interfere with other registers/addresses that are used by other gadgets.
Shellcode

- Q will produce the shellcode, so an exploit can be crafted by hand now, or...
- Given an unharded exploit, Q can automatically create a hardened exploit using the categories ROP gadgets
Exploit Hardening

Trace Based Analysis

- Q uses trace based analysis to analyze what path an exploit takes through a target program.
- Via symbolic execution creates a logical formula for all inputs that will take the same path.

SMT Solver

- All inputs that take the same path (follow the formula) are path constraints.
- All "ret"s in the gadgets form exploit constraints.
- Using an SMT solver, if it is capable of solving the equation given the path constraints and exploit constraints, an exploit that can bypass DEP and ASLR is produced.
### Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Parameters</th>
<th>Semantic Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOOPG</td>
<td>—</td>
<td>—</td>
<td>Does not change memory or registers</td>
</tr>
<tr>
<td>JUMP</td>
<td>AddrReg</td>
<td>Offset</td>
<td>EIP ← AddrReg + Offset</td>
</tr>
<tr>
<td>MOVEREGG</td>
<td>InReg, OutReg</td>
<td>—</td>
<td>OutReg ← InReg</td>
</tr>
<tr>
<td>LOADCONSTG</td>
<td>OutReg, Value</td>
<td>—</td>
<td>OutReg ← Value</td>
</tr>
<tr>
<td>ARITHMETICG</td>
<td>InReg1, InReg2, OutReg</td>
<td>◊b</td>
<td>OutReg ← InReg1 ◊b InReg2</td>
</tr>
<tr>
<td>LOADMEMG</td>
<td>AddrReg, OutReg</td>
<td># Bytes, Offset</td>
<td>OutReg ← M[AddrReg + Offset]</td>
</tr>
<tr>
<td>STOREMEMG</td>
<td>AddrReg, InReg</td>
<td># Bytes, Offset</td>
<td>M[AddrReg + Offset] ← InReg</td>
</tr>
<tr>
<td>ARITHMETICLOADG</td>
<td>OutReg, AddrReg</td>
<td># Bytes, Offset, ◊b</td>
<td>OutReg ◊b ← M[AddrReg + Offset]</td>
</tr>
<tr>
<td>ARITHMETICSTOREG</td>
<td>InReg, AddrReg</td>
<td># Bytes, Offset, ◊b</td>
<td>M[AddrReg + Offset] ◊b ← InReg</td>
</tr>
</tbody>
</table>

Credit: [Exploit Hardening Made Easy, USENIX Security 2011]

**9 out of 9**

- Q was able to successfully harden 9 out of 9 real exploits downloaded from exploit-db
- Completing each one in only a couple of minutes
Final Thoughts

Limitation

- Q uses trace based analysis, so is it possible there are other paths that could be taken to hit a vulnerability, and other payloads that would work
  - Q will not find exploits

Conclusion

- Q automatically generates ROP payloads and hardens exploits
- 20 KB of unrandomized code in an application is enough to complete negate DEP and ASLR