ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks

Ivan Fratrić
University of Zagreb
Faculty of Electrical Engineering and Computing

Zagreb, 24.09.2012
Overview

- Introduction
  - What is a memory corruption vulnerability?
  - Buffer overflow example
- Introduction to return-oriented programming (ROP)
- Related work
- ROPGuard
  - Main ideas
  - Selected Implementation details
  - Evaluation
- Conclusion and ideas for future work
Introduction

- Memory corruption vulnerability
  - contents of a memory location are unintentionally modified due to programming errors

**CVE-2012-4969**

**Summary:** Use-after-free vulnerability in the CMshtmlEd::Exec function in mshtml.dll in Microsoft Internet Explorer 6 through 9 allows remote attackers to execute arbitrary code via a crafted web site, as exploited in the wild in September 2012.

**Published:** 09/18/2012

**CVSS Severity:** 9.3 (HIGH)

**CVE-2012-4166**

**Summary:** Adobe Flash Player before 10.3.183.23 and 11.x before 11.4.402.265 on Windows and Mac OS X, before 10.3.183.23 and 11.x before 11.2.202.258 on Linux, before 11.1.111.16 on Android 2.x and 3.x, and before 11.1.115.17 on Android 4.x; Adobe AIR before 3.4.0.2540; and Adobe AIR SDK before 3.4.0.2540 allow attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2012-4163, CVE-2012-4164, and CVE-2012-4165.

**Published:** 08/21/2012

**CVSS Severity:** 10.0 (HIGH)

**CVE-2012-2524**

**Summary:** Microsoft Office 2007 SP2 and SP3 and 2010 SP1 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted Computer Graphics Metafile (CGM) file, aka "CGM File Format Memory Corruption Vulnerability."

**Published:** 08/15/2012

**CVSS Severity:** 9.3 (HIGH)

- In many cases memory corruption vulnerabilities can lead to arbitrary code execution
A long time ago in a galaxy far, far away....
Example: Buffer overflow on stack

```c
#include <stdio.h>

void main()
{
    char buffer[20];
    gets(buffer);
    ...
}
```
Example: Buffer overflow on stack

```c
#include <stdio.h>

void main()
{
    char buffer[20];
    gets(buffer);
    ...
}
```

- Stack growth
  - Frame of `main()`
    - `char buffer[20]`
    - Frame pointer
    - Return address
  - Function arguments
  - Local variables
  - Frame pointer
  - Return address
  - ...
  - Frame of another function
Example: Buffer overflow on stack

```c
#include <stdio.h>
void main()
{
    char buffer[20];
    gets(buffer);
    ...
}
```

When `main()` returns, the attacker gains control over control flow (EIP)
Example: Buffer overflow on stack
Memory corruption vulnerabilities

- Many additional details about stack buffer overflows
  - Stack cookies, SEH overwrite, SafeSEH, SEHOP
- Many other memory corruption vulnerabilities
  - Heap overflow
  - Integer overflow
  - Use-after-free
  - Double free
  - Format string vulnerabilities
  - Improper bound checks
  - Improper loop conditions
  - Etc.

In common: Attacker gains control of EIP and can execute arbitrary code
Data Execution Prevention (DEP)

- Hardware protection against exploitation
- A special flag (NX bit) indicates executable memory regions
  - Executable modules loaded in memory (.exe, .dll, etc.) are executable
  - Stack and heap are NOT executable
    - Can be made executable by calling special function i.e. VirtualProtect()
- Introduced on Linux in kernel 2.6.8, on Windows in Windows XP Service Pack 2
Return-oriented programming

- Generalization of return-to-libc and similar attacks
- Use small pieces of existing executable code to perform (complex) actions specified by the attacker
  - “small pieces of existing executable code” are called gadgets
Return-Oriented Programming is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.
Return-oriented programming

- **Gadget consists of two parts:**
  - Instruction(s) that perform something useful
  - A part that transfers the code execution to the next gadget

- **RETN instruction**
  - Can be used to transfer execution to the next gadget if the attacker controls the stack
Return-oriented programming

- Simple example:
  - Attacker wants to write value 0x00001337 to address 0x12345678

- Break it into simple operations so that we can find appropriate gadgets in executable modules
  - Load 0x1337 into EAX
  - Load 0x12345678 into ECX
  - Do MOV [ECX],EAX
Return-oriented programming

- Simple example (cont.)
  - Attacker wants to write value 0x00001337 to address 0x12345678

- See if we have appropriate gadgets in executable code

- msvcr71.dll:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>7C344CC1</td>
<td>58</td>
<td>POP EAX</td>
</tr>
<tr>
<td>7C344CC2</td>
<td>C3</td>
<td>RETN</td>
</tr>
<tr>
<td>7C3410C3</td>
<td>59</td>
<td>POP ECX</td>
</tr>
<tr>
<td>7C3410C4</td>
<td>C3</td>
<td>RETN</td>
</tr>
<tr>
<td>7C3503C8</td>
<td>8901</td>
<td>MOV DWORD PTR DS:[ECX],EAX</td>
</tr>
<tr>
<td>7C3503CA</td>
<td>C3</td>
<td>RETN</td>
</tr>
</tbody>
</table>
Return-oriented programming

- Simple example (cont.)
  - Attacker wants to write value 0x1337 to address 0x12345678
- Putting it all together

```
EIP
???????? RETN

7C344CC1 POP EAX
7C344CC2 RETN

7C3410C3 POP ECX
7C3410C4 RETN

7C3503C8 MOV [ECX], EAX
7C3503CA RETN

ESP
0x7C344CC1
0x00001337
0x7C3410C3
0x12345678
0x7C3503C8
0x????????
```

EAX: 00001337
ECX: 12345678
Return-oriented programming

- Real-world example

```plaintext
0x7c37653d,  # POP EAX  # POP EDI  # POP ESI  # POP EBX  # POP EBP  # RETN
0xffffffffdffe,  # Value to negate, will become 0x000000201 (dwSize)
0x7c347f98,  # RETN (ROP NOP) [msvcr71.dll]
0x7c3415a2,  # JMP [EAX] [msvcr71.dll]
0xffffffffffe,  #
0x7c376402,  # skip 4 bytes [msvcr71.dll]
0x7c351e05,  # NEG EAX  # RETN [msvcr71.dll]
0x7c345255,  # INC EBX  # FPATAN  # RETN [msvcr71.dll]
0x7c352174,  # ADD EBX,EAX  # XOR EAX,EAX  # INC EAX  # RETN [msvcr71.dll]
0x7c344f87,  # POP EDX  # RETN [msvcr71.dll]
0xffffffffc0,  # Value to negate, will become 0x00000040
0x7c351eb1,  # NEG EDX  # RETN [msvcr71.dll]
0x7c34d201,  # POP ECX  # RETN [msvcr71.dll]
0x7c38b001,  # &Writable location [msvcr71.dll]
0x7c347f97,  # POP EAX  # RETN [msvcr71.dll]
0x7c37a151,  # ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll]
0x7c378c81,  # PUSHAD  # ADD AL,0EF  # RETN [msvcr71.dll]
0x7c345c30,  # ptr to 'push esp  # ret ' [msvcr71.dll]
```
Return-oriented programming

- Unintended instruction sequences
  - Example:
    
    | Address   | Opcode  | operand description |
    |-----------|---------|---------------------|
    | 7C346C09  | 0F58C3  | ADDPS XMM0, XMM3    |
    | 7C346C0A  | 58      | POP EAX             |
    | 7C346C0B  | C3      | RETN                |

- Other instructions can be used to connect gadgets instead of RETN:
  - Indirect jumps (jump-oriented programming, JOP)
    - JMP EAX
    - JMP [EAX]
    - JMP [EAX + offset]
  - Indirect calls
The unexpected twist

ROP is Turing-complete (Shacham, 2007)

No! That's not true! That's impossible!
Mitigations (related work)

- **Address Space Layout Randomization (ASLR)**
  - Randomizes base address of
    - Executable modules
    - Stack
    - Heap
    - etc.

- **Can be bypassed by**
  - Using/loading a module that does not support ASLR
  - Using a secondary vulnerability to perform memory disclosure
  - Using the same memory corruption vulnerability to perform both memory disclosure and code execution
    - Example: Memory disclosure technique for Internet Explorer
Mitigations (related work)

- Solutions based on dynamic binary instrumentation
- ROPdefender (Davi et al., 2011)
  - “Shadow stack” approach
  - CALL-RETN relations (ROP: RETN without appropriate CALL)
    - On each CALL, the return address is placed on a shadow stack along with the “real” stack
    - On each RETN, we check if the address on top of the stack is the same as the address on top of the shadow stack
- Drawbacks
  - Dynamic instrumentation introduces overhead of 2x
  - Protects only against RETN-based gadgets
Mitigations (related work)

- Compiler-level approaches

- G-Free (Onarlioglu et al., 2009)
  - Removes all unintended gadgets
  - “Encrypts” return addresses in function prologue and “decrypts” before the function ends
  - Adds stack cookie to all functions with indirect jumps/calls. The cookie is checked before the jump/call is made

- Comprehensive solution, but:
  - Requires knowing the source code
  - Needs to be applied to all modules in order to be effective
Mitigations (related work)

- Static binary rewriting
- In-Place Code Randomization (Pappas et al., 2012)
  - Changes the order of instructions
  - Replaces instructions with equivalent ones
- Drawbacks
  - Relies on automated disassembly
    - Not an exact science!
    - Code vs. data
    - Indirect call/jump targets
ROPGuard: main idea

- Requirements:
  - Prototype must be fully functioning and work on Windows
  - Prototype must have low overhead meaning CPU and memory cost of no more than 5%
  - Prototype must not have any application compatibility or usability regressions

- Can we avoid instrumentation/recompiling/rewriting by using the information already present in the process?

- Design practical runtime checks that can be applied at runtime

- When to perform the checks?
In order to leverage the attack, the attacker will need to call some functions (critical functions) to escape the constraints of ROP or current process:

- VirtualProtect, VirtualAlloc, LoadLibrary
- CreateProcess
- OpenFile, WriteFile
- Etc.

Critical functions and their locations:

CriticalFunction = kernel32.dll:VirtualProtect:4
CriticalFunction = kernel32.dll:VirtualProtectEx:5
CriticalFunction = kernel32.dll:VirtualAlloc:4
CriticalFunction = kernel32.dll:VirtualAllocEx:5
CriticalFunction = kernel32.dll:HeapCreate:3
CriticalFunction = ntdll.dll:RtlCreateHeap:6
CriticalFunction = kernel32.dll:CreateProcessA:10
CriticalFunction = kernel32.dll:CreateProcessW:10
CriticalFunction = kernel32.dll:CreateProcessInternalA:12
CriticalFunction = kernel32.dll:CreateProcessInternalW:12
CriticalFunction = kernel32.dll:LoadLibraryA:1
CriticalFunction = kernel32.dll:LoadLibraryW:1
CriticalFunction = kernel32.dll:LoadLibraryExA:3
CriticalFunction = kernel32.dll:LoadLibraryExW:3
CriticalFunction = kernel32.dll:CreateRemoteThread:7
CriticalFunction = kernel32.dll:WriteProcessMemory:5

# Filesystem functions
CriticalFunction = kernel32.dll:CreateFileA:7
CriticalFunction = kernel32.dll:CreateFileW:7
CriticalFunction = kernel32.dll:WriteFile:5
CriticalFunction = kernel32.dll:WriteFileEx:5

# Registry functions
CriticalFunction = advapi32.dll:RegOpenKeyA:3
CriticalFunction = advapi32.dll:RegOpenKeyW:3
CriticalFunction = advapi32.dll:RegOpenKeyExA:5
CriticalFunction = advapi32.dll:RegOpenKeyExW:5
CriticalFunction = advapi32.dll:RegCreateKeyA:3
CriticalFunction = advapi32.dll:RegCreateKeyW:3
CriticalFunction = advapi32.dll:RegCreateKeyExA:9
CriticalFunction = advapi32.dll:RegCreateKeyExW:9
CriticalFunction = advapi32.dll:RegSetValueA:5
CriticalFunction = advapi32.dll:RegSetValueW:5
CriticalFunction = advapi32.dll:RegSetValueExA:6
CriticalFunction = advapi32.dll:RegSetValueExW:6

Etc.
ROPGuard: main idea

- Perform runtime checks when any critical function gets called
- Attempt to answer questions
  - How did the critical function get called?
  - What will happen after the critical function executes?
  - Is the current state of the system consistent with the normal program execution or with the exploit attempt?
  - Will executing the critical function violate the system’s security?
- ROPGuard defines 6 runtime checks
ROPGuard: runtime checks(1)

- Check the stack pointer
- Assume: Attacker controls EIP and EAX, but not the stack
  - Stack pivoting

- Thread information block contains information about the area of the memory that was designated for the stack when the thread was created

```
MOV ESP, EAX
RETN

XCHG EAX, ESP
RETN
```
ROPGuard: runtime checks (2)

- Look for the address of critical function above the top of the stack
- Why?
  - RETN:
    - EIP <- ESP
    - ESP <- ESP+4
  - If we entered critical function via RETN, the address of critical function must be just above the top of the stack
- ROPGuard “saves” a part of the stack upon entering the critical function for examination
ROPGuard: runtime checks (3)

- Return address check

- For each critical function, verify that
  - The return address is executable
  - The instruction at the return address must be preceded with a CALL instruction
  - CALL instruction must lead back to the current critical function
Check the call stack
  - Call stack must be valid

How do we obtain call stack?

Before RETN

```
mov esp, ebp;
pop ebp;
```

Return address just below the frame pointer!
ROPGuard: runtime checks (4)

- Checking the call stack using frame pointers

```c
frame_ptr = EBP;
for a specified number of frames
    check if frame_ptr points to the stack;
    return address <- [frame_ptr + 4];
    check if return address is executable;
    check if return address is preceded by call;
frame_ptr = [frame_ptr];
```
ROPGuard: runtime checks (4)

- Checking the call stack using frame pointers

**Drawbacks**

- Compilers are not required to use frame pointers!
- Sometimes a compiler will opt to omit frame pointer in favor of using EBP as an additional general-purpose register
- Frame pointers are generally not used for very short functions
- Can be regulated by a compiler switch
ROPGuard: runtime checks(5)

- Can we walk the call stack without relying on frame pointers?
- Can we determine the size of the stack frame by relying only on the machine code?

EIP -> 7C914EEE  MOV AX,WORD PTR DS:[ESI]
ESP = ESP + 12 -> 7C914EF1  ADD ESP,0C
              7C914EF4  CMP AX,WORD PTR DS:[ESI+2]
              7C914EF8  JNB SHORT ntdll.7C914F01
              7C914EFA  SHR EDI,1
              7C914EFC  AND WORD PTR DS:[EBX+EDI*2],0
ESP = ESP + 4  -> 7C914F01  POP EBX
              7C914F02  XOR EAX,EAX
ESP = ESP + 4  -> 7C914F04  POP EDI
ESP = ESP + 4  -> 7C914F05  POP ESI
RETURN ADDRESS = [ESP] -> 7C914F07  RETN
ROPGuard: runtime checks (5)

- ROPGuard simulates control flow from return address of the critical function to the next return instruction and keeps track of ESP along the way
  - Repeat from the return address

- Potential problems
  - Stack frame determined dynamically
    - Very rare in practice
  - stdcall calling convention in combination with
  - Indirect calls: CALL EAX; CALL [EAX] etc.
ROPGuard: runtime checks (5)

- ROPGuard brakes simulation when it reaches an instruction for which it cannot resolve ESP
- Possible extension: simulate entire instruction set
- For the time being:

```assembly
0x7c37653d, # POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
0xffffffffdf, # Value to negate, will become 0x00000201 (dwSize)
0x7c347f98, # RETN (ROP NOP) [msvcr71.dll]
0x7c3415a2, # JMP [EAX] [msvcr71.dll]
0xffffffff, #
0x7c376402, # skip 4 bytes [msvcr71.dll]
0x7c351e05, # NEG EAX # RETN [msvcr71.dll]
0x7c345255, # INC EBX # FPATAN # RETN [msvcr71.dll]
0x7c352174, # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
0x7c344f87, # POP EDX # RETN [msvcr71.dll]
0xfffffffff0, # Value to negate, will become 0x00000040
0x7c351eb1, # NEG EDX # RETN [msvcr71.dll]
0x7c34d201, # POP ECX # RETN [msvcr71.dll]
0x7c38b001, # &Writable location [msvcr71.dll]
0x7c347f97, # POP EAX # RETN [msvcr71.dll]
0x7c37a51, # ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll]
0x7c378c81, # PUSHAD # ADD AL,0EF # RETN [msvcr71.dll]
0x7c345c30, # ptr to 'push esp # ret ' [msvcr71.dll]
```
ROPGuard: runtime checks (6)

- Function-specific checks
  - Do not allow program to make stack executable
  - Do not allow program to load .dll-s from the network
**ROPGuard: Implementation details**

- ROPGuard is implemented as a command line tool and a .dll
- Process is started in a suspended state
- dll injection via CreateRemoteThread()
- When the dll is loaded
  - Hooks all critical function to perform appropriate checks using inline hooking
  - Function header is replaced with a direct jump to

```assembly
SUB ESP, PRESERVE_STACK; //save part of the stack for later examination
PUSHAD; //save the state of all registers at the moment of function call
PUSH ESP; //pointer to the stored registers array
PUSH ORIGINAL_FUNTION ADDRESS; //address of the current critical function
CALL RopCheck; //perform the appropriate checks
ADD ESP, PRESERVE_STACK+32; //restore the stack pointer
//resume normal function execution
[original function header]
JMP ORIGINAL_FUNTION_ADDRESS + size of original function header;
```
ROPGuard: Implementation details

- Whenever a process creates another (child) process, dll is injected into this process as well.
- Cache information about executable module (avoids repeated calls to VirtualQuery).
- ROPGuard can be used to protect processes that are already running.
- Extensive configuration options:
  - Define what checks to perform
  - Define critical functions
ROPGuard: Evaluation

- Experiments on an example vulnerable application
**ROPGuard: Evaluation**

- A series of benchmarks was performed to determine the computing overhead

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Benchmark type</th>
<th>Score, not protected</th>
<th>Protected, no cache</th>
<th>Protected, with cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCMark Vantage</td>
<td>System</td>
<td>5049</td>
<td>5009</td>
<td>0.80%</td>
</tr>
<tr>
<td>NovaBench</td>
<td>System</td>
<td>799</td>
<td>784</td>
<td>1.91%</td>
</tr>
<tr>
<td>Peacekeeper</td>
<td>Browser</td>
<td>1480</td>
<td>1406</td>
<td>5.26%</td>
</tr>
<tr>
<td>SunSpider</td>
<td>Browser</td>
<td>247.7 s</td>
<td>253.8 s</td>
<td>2.46%</td>
</tr>
<tr>
<td>3DMark06</td>
<td>Gaming</td>
<td>7994</td>
<td>7992</td>
<td>0.03%</td>
</tr>
<tr>
<td>SuperPI 16M</td>
<td>CPU</td>
<td>403.0 s</td>
<td>399.5 s</td>
<td>-0.87%</td>
</tr>
</tbody>
</table>

**Average overhead**

- 0 false positives while running the benchmarks with the default configuration.
ROPGuard: Evaluation

- ROPGuard .dll is just 48kB in size.
- Additional memory overhead introduced by copy-on-write memory page protection
ROPGuard: Evaluation

- ROPGuard won the second prize in Microsoft’s BlueHat Prize contest at Black Hat USA 2012
ROPGuard: Evaluation

- ROPGuard has been integrated with Microsoft’s EMET tool
  - Enhanced Mitigation Experience Toolkit
Conclusion

- Preventing ROP is a difficult problem
  - Still largely unsolved!

- ROPGuard
  - Can detect currently used ROP attacks
    - Raises the bar for the attacker, more costly exploit development
  - Easy to deploy to protect existing programs
  - Low CPU and memory overhead

- Source code and documentation available at
  http://code.google.com/p/ropguard/
Ideas for future contests

- Contest evaluation criteria
  - 40.00% - Impact (Strongly mitigate modern threats?)
  - 30.00% - Robustness (Easy to bypass?)
  - 30.00% - Practical and Functional

- Find ways to improve the reliability of binary rewriting
  - Modify binary without breaking basic blocks
    - Removal of unintended gadgets
    - Binary modification relying on unintended instruction sequences
  - Code randomization
    - Resolve code-vs-data and basic blocks dilemma by running the original binary
    - On the first run, the code is modified, later only the modified code is run
Other contest finalists

- **KBouncer (V. Pappas, 2012)**
  - Recent Intel CPUs support Last Branch Recording (LBR)
  - Stores the last branches in a set of 16 model specific registers (MSRs), can be read using `rdmsr` instruction
    - `Recordv` only return instructions
    - On every system call check if call instruction precedes the return address
Other contest finalists

/ROP (J. DeMott, 2012)

- Compiler-level solution
- Makes a list of valid return addresses
- Requires interrupt on each return instruction
  - Check if the return address is in the whitelist
RPGuard: runtime checks(5)

EIP = return address of critical function;
for a specified number of instructions
  decode instruction at [EIP];
  update EIP;
if current instruction changes ESP
  update ESP;
else if current instruction is RETN
  check if return address is executable;
  check if return address is preceded by call;
else if current instruction changes ESP in an unresolvable way
  break simulation;