1 Contents

2 Executive Summary

This novel defensive lowers the effect of address space disclosures and mitigates ROP exploits, by checking that the target address of every return instruction is safe. This works because known ROP exploits contain return instructions. I call this protection /ROP because it integrates with Windows and Visual Studio similar to other Microsoft brand name protections such as /DynamicBase, /NXCompat, /GS, etc. Similar to those protections, /ROP has relatively low performance impact and is practical to be rolled out on all modules across all Windows platforms. /ROP is difficult to bypass; creating an immediate positive impact on the security posture of all deployed Windows systems.

NOTE: This submission was done on my own time and is in no way affiliated with my current employer, or a reflection of their views or technical capabilities.

3 Introduction

Address randomization (ASLR) and page marking (DEP) are two recent Microsoft protections designed to defeat memory corruption bugs and the resultant control flow hijacks, which allows arbitrary code execution. The concept is solid: If an attacker cannot execute arbitrary code in data memory, most attacks of this nature are foiled. DEP addresses this issue. However, attackers learned that by bouncing through DLLs to form a new code pattern, page protections could be changed (or new attacker friendly pages allocated) to allow execution in memory. ASLR addresses this issue by changing the base address of each module, making it hard for attackers to know where the desired code patterns reside.

4 Problem: ROP

One of the largest current threats to compiled software is Return-Oriented Programming (ROP). Why does ASLR and DEP not prevent ROP as it's supposed to? In some cases it does. In many cases it does not, for one or more of the following reasons:

1. Unprotected modules
    - Visual Studio protections need to be compiled in, and older versions of programs which are still in use, might not yet have been updated.
    - As an example, recent versions of Java shipped with an old version of the Microsoft Visual C++ Run-Time, a module named MSVCR71.dll. MSVCR71.dll is not compiled with DEP/ASLR. Thus, an attacker can leverage that module by forcing it into the current address space and "ROPing" through gadgets to disable DEP.
    - The misuse of MSVCR71.dll is common enough that a payload called a universal DEP/ASLR bypass has been created by researchers at the Corelan security group[1].
2. Address space information disclosures
    - Modern applications, such as web browsers, present a large attack surface for attackers to search for bugs. Underlying components of feature rich applications, such as scripting languages, often provide bugs such as use-after-frees, object confusion, etc. which an attacker can leverage to leak the location of internal memory. The base address of modules can be discovered via the leak. Next the attacker needs a multi-purpose bug, or another bug to corrupt memory and complete the ASLR/DEP bypass.
3. Novel Techniques
    - If history has shown us anything, it is that attackers will continue to find novel ways to bypass current protections.

5 Solution: /ROP

The provided prototype will show that ROP attacks can be stopped by determining if the return instruction[2] points to a safe address. Filtering return addresses could either be done with a black or whitelist. Whitelisting is known to be a better security solution. This has been shown to be true in IP

[1] https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/
[2] RET or RETN are the assembly mnemonics for a return instruction in the x86 architecture.

address filtering, input sanitization, and more. Thus, we create a list of known safe addresses for which return instructions are allowed to transfer the instruction pointer to.

5.1 Differentiator

The proposed whitelisting technique is superior to prior techniques for two reasons: First, it is hardware assisted which means that applications do not need to be instrumented in a dynamic binary instrumentation (DBI) framework[3]. Instrumentation is too slow for practical applications. Second, our technique enforces correctness of intended *and unintended* return instructions throughout all code modules. Unintended instructions are ad hoc instructions which can be located by searching the middle opcode bytes of intended instructions in x86. Related works will be detailed at the end of the document.

5.2 /ROP High Level Algorithm

The following pseudocode describes the hardware enforced security check, which would be performed before every intended or unintended return:

```
if( return_instruction)
{
        if ( Check_WhiteList () )
        {
                return();
        }
        else
                report_ROP_failure(BAD_RET_ADDR_ERROR);
}
```

For each module, a whitelist of safe return locations is created. That list is distributed as a section inside the code module. As a process is loaded, and code modules are loaded into memory, each whitelist is added to a master whitelist in memory[4]. This technique depends on cooperation with CPU makers[5]. A software interrupt needs to be triggered whenever a return instruction is executed. The whitelist is checked. If the check succeeds, the return is allowed. If not, the process fails safe with a controlled shutdown, similar to other security failure checks.

5.3 Windows and Visual Studio Integration

The /ROP protection is intended to be compiled into each module, similar to other Microsoft protections, such as /DynamicBase (ASLR), /NXCompat (DEP), /GS (Stack Cookies), /SafeSEH (Safe Exception Handling), etc. Below is the design specification for the new /ROP protection:

[3] Dynamic binary instrumentation techniques are used to execute additional instructions at certain locations in a program.

[4] The whitelist would need to be randomized via ASLR so that attackers could not change it before launching a ROP attack with a wild write vulnerability. Otherwise it could be place in kernel memory if desired.

[5] This new CPU feature is simple. It is in line with what semiconductor chip makers AMD and Intel did when they provided the security feature referred to as the Non-Executable Bit (NX/XD).

1. The next Visual Studio release would compile modules to have /ROP by default, thus only a /NOROP flag is required if the protection is to be manually disabled by developers. Even if a CPU does not support /ROP, the whitelist embedded in modules will cause no ill effects.
2. Visual Studio builds a table of addresses for the given module of safe return locations. A safe address is one that is:
    a. Directly after a function[6], and is thus a valid return target. All other locations are not in the whitelist, and are therefore unsafe to return to.
        i. Note: The process of identifying these locations without source code is error prone. This is discussed later.
    b. Each address is actually an offset that will be placed in the .ROP section within each compiled module. The offsets could be fixed width binary numbers. The prototype uses ASCII numbers separated by linefeeds for simplicity.
3. As Windows is loading each module into a processes address space, a list of offsets per module is collected into a master list of allowed addresses.
    4. Windows code must be added to implement the *Check_WhiteList ()* routine. This code is demonstrated for x86 in the prototype code, and by the pseudocode below:

```
Check_WhiteList()
{
        set<ADDRINT>::iterator set_it;
        map<string, set<ADDRINT> >::iterator it;
        bool found = FALSE;
        unsigned int ret_target;
        _asm { lea ret_target, [ESP] }

        for( it=whitelist.begin(); it != whitelist.end(); it++)
        {
            set_it = it->second.find(ret_target);

            if( set_it != it->second.end() )
            {
                found = TRUE;
                break;
            }
        }
      return found;
}
```

5. Each time the processor encounters a return instruction, Windows code runs and searches the whitelist to see if the return target is allowed.
6. The program will fail safe (crash, as it does with other protections) if the address is not safe, otherwise the program continues execution as normal.

[6] A function is represented by a CALL assembly instruction in x86.

6 Prototype

The core of the prototype is written in C++ and leverages the Pin DBI [Pin 2011]. Instrumentation is used to simulate hardware support of return instruction interrupts as described previously. The process to identify safe return locations within executable code (without source code) is done either with Pin, IDA pro [Hex-Rays 2012], or both. Another Pintool is used to load and check the whitelist, as would be done by Windows. Python and batch scripts are used to automate the tests to collect memory and CPU performance metrics. All this code (and more) is contained in the provided *prototype.zip* file. The zip file contains two directories: *ropstop* and *stackoverflow*. The *ropstop* directory contains the prototype code and various python and batch scripts to test it. The *stackoverflow* directory contains a sample application.

6.1 Prototype Operation Instructions

The prototype is broken into three Pintool components: compiler, exception, and loader. The compiler is a Pintool that simulates the creation of the .ROP section as would be done by Visual Studio. The exception Pintool will not be required for Microsoft since source code would be available. However, it is required for the prototype since correctly finding all the safe return locations is error prone via static disassembly. Runtime information is used to augment static analysis. The loader Pintool is used to simulate Windows loading, executing, and monitoring a process. As return instructions are encountered, the master whitelist is consulted, and the process either continues, or in the case of a whitelist miss, fails with a ROP security error. The next two sections provide step-by-step instructions.

6.1.1 Running the Prototype against the Included Sample Application

In the *stackoverflow* directory is the source code and build script for a simple application which contains a stack buffer overflow. The /DynamicBase, /NXCompat, and /GS protections are disabled to show that the new /ROP protection can stop ROP attacks, and in fact regular stack overflows[7] without depending upon other protections.

The following steps[8] show how to build, use, and test the performance of /ROP against this simple application called *test.exe*:

1. Install Microsoft Visual studio.
2. Install the Intel Pin DBI package[9].
3. Unzip the *prototype.zip* file. Place the *ropstop* and *stackoverflow* directories in the *..\Pin\source\tools\* directory.
4. Open a command shell and change directory to the *ropstop* directory.
5. **Run** the following command: *..\..\..\pin –help*. The version tested reads: *Pin 2.10 VERSION: 45445 BUILDER: BUILDER DATE: Nov 22 2011*.
   - *Test.exe* in the *stackoverflow* directory is already built, but you can rebuild by executing the *build.bat* script if desired.

[7] Normal stack overflows also return to locations that will not be in the whitelist, and will be mitigated.
[8] Where ever you see the word "run" in bold, type and execute a command in a Windows command shell if you desire to follow along.
[9] http://www.pintool.org/downloads.html. Be sure to get the version that matches your Visual Studio version.

- Each Pintool can be compiled separately: *c_build.bat* compiles *compiler.cpp*; *e_build.bat* compiles *except.cpp*; *l_build.bat* compiles *loader.cpp*. However, there is a *MASTER_TEST.py* script which can be given a *–c* flag, and it will compile all three. You must have python[10] installed.
- Each of the Pintools can be run individually with a variety of options that are documented within each program. The usage is printed to the screen for all command line programs, if an incorrect parameter is used. For the test program, and for the Internet Explorer example discussed in the next section, there exist batch scripts to assist with parameters. For example, the *l_run_TEST.bat* is the batch script with runs the *test.exe* program with the compiled *loader.dll* Pintool. Internally the batch script looks like:
  ```
  ..\..\..\pin -t obj-ia32\loader.dll -t -a -o loader_TEST.txt -- ..\stackoverflow\test.exe attack.bin
  ```
  In English, that means:
    - Instrument via Pin with our compiled loader.dll Pintool for 32bit Windows
    - –t: /ROP security enabled
    - –a: Uses all of the corresponding *module.rop* files as the .ROP sections
    - –o: The log file to output trace information to
    - -- is the pin directive indicating that the name of the executable to instrument follows
    - *..\stackoverflow\test.exe* is the relative path to the test program
    - *attack.bin* is the input file consumed by our test program
- The CPU performance of the /ROP protection on the test application is measured with the *check_CPU_performance.py* script. Likewise, the memory usage is checked via the *check_MEM_performance.py*. This can be done individually.

6. However, the first command you should now **run** is the master[11] script for the test program as shown below:

   ```
   python Master_TEST.py –c
   ```
   - The *–c* compiles all three Pintools automatically.
   - Section 6.2.2 discusses the performance results from the master scripts.
7. From the *ropstop* directory, **run** the following two commands to examine the ROP exploit for *test.exe*:

   ```
   ..\stackoverflow\test.exe test.bin
           one 1 1
           default
           Calling vuln()
           Called vlun()
   ..\stackoverflow\test.exe attack.bin
           one 1 1
           default
           Calling vuln()
           You found the secret function!
   ```

[10] My version was 2.7.1.
[11] A master script is one that calls a series of scripts and commands to run a program through a sequence of tests.

We see that the test program called with the *test.bin* input enters and exits a vulnerable function without exploiting the bug. If the input *attack.bin* is supplied, a ROP exploit runs and an unintended function is executed.

8. **Run** the *l_run_TEST.bat* script to demonstrates that the /ROP protection mitigates this attack. The output appears as follows:

```
..\..\..\pin -t obj-ia32\loader.dll -t -a -o loader_TEST.txt -- ..\stackoverflow\test.exe attack.bin
Protecting: test.exe
Protecting: apphelp.dll
Protecting: KERNELBASE.dll
Protecting: kernel32.dll
Protecting: ntdll.dll
Calling vuln()
[!] Found UNSAFE RET: test.exe+1000
EXITING to prevent ROP exploits
```

The exploit overwrites the stored return address on the stack and attempts to return directly to the function at address *test.exe+1000*[12]. But since this function is never legitimately used in the program, returning to this function is not in the whitelist. /ROP correctly denies this return.

6.1.2 Testing the Prototype with MS11_050

To test the effectiveness of the approach against a real world exploit, an Internet Explorer vulnerability is used. MS11_050 is a use-after-free bug in the mshtml code module. Metasploit has an exploit for MS11_050. Metasploit is one of the most widely used penetration testing tools, and it is freely available. It is likely that malware authors base their exploits off Metasploit modules. Certainly other private exploits exist. The results shown here would work for public or private ROP exploits.

The following is the list of steps required to test the /ROP protection using a MS11_050 exploit:

1. Set up a 32bit Windows 7[13] VM with IE8. Install ruby for Windows[14]. Install Java 6 update 27[15] for x86 Windows. Configure the default page in IE to open to http://localhost/exploit.html. Double click *ie8_single_process.reg* in the *ropstop* directory to set IE to be a single process, to ease testing. Disable UACs to ease setup and testing.
   - Building the .ROP files for each of the many DLLs used by IE takes time and the *compile.dll* Pintool fails on some modules. Because of that, two things are done here to make this demonstration simpler. First, I have included the .rop file for MSVCR71.dll[16], which is the key file that this exploit takes advantage of. That allows the reader of this document to test the prototype without additional requirements. Second, I have a more advanced IDA pro python script that more accurately statically analyzes the required DLLs. I have included a document in

[12] This function is called jackpot(). Review the .\stackoverflow\test.cpp source code file for complete details.
[13] Service Pack 1, but obviously not fully patched. The vulnerability in IE needs to exist for the test to work.
[14] http://rubyinstaller.org/downloads/
[15] http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html#jdk-6u27-oth-JPR
[16] It is in the .*ropstop**rop*\ directory. Be sure not to overwrite this file before doing this test.

the *ropstop\doc* directory called *Complete_IE_Testing_guide.docx* which gives detailed instructions on how to fully test the /ROP prototype across all loaded modules. Viewing that document is not required nor part of this submission.

2. For this demonstration, open two command shells in the VM.
   a. In the first window, **run** *ruby MS11_050.rb* from the *ropstop* directory. I have extracted the Metasploit exploit into a stand-alone ruby file to ease step up.
   b. In the second window, **run** *l_run_IEXPLORE_just_MSVCR71.bat*. IE will open, but then close. It closes, because the /ROP protection stops IE from returning to an address within MSVCR71.dll that is not in the whitelist. Open the *loader_IEXPLORE.txt* to view run details. The last few lines of the file are:

```
7c348b05: xchg esp, eax
7c348b06: ret
[!] Found UNSAFE RET: msvcr71.dll+36402
```

The exploit demonstrated here attempts to return to *msvcr71.dll+36402*[17] after doing a ROP technique call a stack flip. The return address points to a gadget that pops the EBP register. Shown from IDA pro:

```
.text:7C376402        pop    ebp       ; Tries to return here after the stack flip.  /ROP blocks it.
.text:7C376403        retn
```

In this example, we saw that the first return instruction that points to an address not within the whitelist is correctly denied.

6.2 Practical and Functional

Building /ROP will require three changes:

1. The Visual Studio compiler will need to be updated to output whitelist entries into the new .ROP section for each module.
2. CPU manufactures will need a way to signal Windows when a return instruction has been executed.
3. Windows will need to be updated to check the whitelist when it receives the signal that a return instruction is being attempted.

These changes are strait forward and in line with other changes that were required for other exploit mitigations such as /DynamicBase, /NXCompat, /GS, and /SafeSEH.

6.2.1 Large Scale Deployment

Performing the /ROP security check on all modules for all applications is practical and feasible with a normal amount of new feature effort. The only major hurdle is to get chip manufactures on board, but considering the importance of security, this should require a relatively small effort.

6.2.2 Low Overhead

From the *ropstop* directory, run the following command: *python Master_ping.py*. This script takes the *ping*[18] program through a series of tests, which measure the effects of the /ROP protect on a real world

[17] The virtual address is 0x7C376402.

program. *Ping* loads a surprising number of DLLs and is therefore a good performance test. The average impact is shown below:

Normal Windows 7 PC:

Overall CPU cost:	+3.29%
Overall Memory cost:	+2.66%

Inside a Windows 7 virtual machine (VM):

Overall CPU cost:	+3.52%
Overall Memory cost:	+3.14%

The CPU tests vary about 1% between runs. The average comes from 5 sets of 10 tests: *ping* is run 5 times with the protection on, and 5 times with it off. The average is calculated. This is done 5 total times. The average shown is the overall average. The memory increase should be the same each time. The memory increase is the number of bytes in the whitelist added to the original module size, plus a fixed amount to account for the size of the new Windows function to check the whitelist[19]. The memory variance between the PC and VM happens because different modules (DLLs) are loaded. Complete details are in the *Performance* and *_performance* directories after running the master script.

6.2.3 Compatibility

The /ROP protection is free from application and usability regressions. That is, if older modules were not compiled with /ROP they can still run with an application that has some /ROP protected modules. However, ROP attacks would be more likely for those modules not protected, and opting-in should be the default in the next Visual Studio release.

7 Related Works and Security Analysis of /ROP

The overall security impact of /ROP is impressive. Previous successful exploits, such as the Metasploit MS11_050 attack, are mitigated as would any other attack that depends on returning to arbitrary locations. Prior defenses have been proposed for ROP that depend on the use of return instructions [Davi et al. 2009; Chen et al. 2009; Francillon et al. 2009; Li et al. 2010; Davi et al. 2011]. None of these are as effective, practical, and easy to implement as the new /ROP defense presented here.

There are two potential weaknesses with /ROP. This first would involve returning to valid addresses within the whitelist. The second would involve not using return instructions at all. The potential for either is low, since the proper gadgets to disable DEP and run a payload are very unlikely to exist under those conditions. We discuss the second weakness in more detail below.

[18] Ping (packet internet groper) is a computer network administration utility used to test the reachability of a host on an Internet Protocol (IP) network and to measure the round-trip time for messages sent from the originating host to a destination computer.
[19] See the associate python scripts (*check_MEM_performance* and *check_CPU_performance*) for complete details.

A new idea for an exploit payload that is completely free from return instructions is called Jump Oriented-Programming (JOP) [Bletsch et al. 2011; Checkoway et al. 2010]. This attack would use indirect jump instructions instead of return instructions. JOP has not been reported in the wild to my knowledge. As a counter measure, the /ROP protection could be updated to whitelist the legitimate target address for jumps. JOP was not considered in this prototype for performance considerations.

Finally, [Erlingsson et al. 2006] claim that the ultimate answer to ROP and JOP would be control-flow integrity (CFI). However, the approach suggested in that paper, known as XFI, suffers from performance and security pitfalls. In particular, unintended returns were not considered in XFI.

8 Summary

The significance of this approach is multi-faceted: First, /ROP is simple to understand and implement. Second, it fits the current Microsoft paradigm. Third, it works with low overhead. Finally, /ROP mitigates all known practical ROP attacks. I hope you will select my technique and add it to the Microsoft suite of exploit mitigation techniques.

9 References

BLETSCH, T., JIANG, X., AND FREEH, V. W. Jump-oriented programming: A new class of code-reuse attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIA CCS), 2011.

CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. 2010. Return-oriented programming without returns. In Proceedings of CCS 2010, A. Keromytis and V. Shmatikov, Eds. ACM Press, 559–72.

CHEN, P., XIAO, H., SHEN, X., YIN, X., MAO, B., AND XIE, L. 2009. DROP: Detecting return-oriented programming malicious code. In Proceedings of ICISS 2009, A. Prakash and I. Sengupta, Eds. LNCS, vol. 5905. Springer-Verlag, 163–77.

DAVI, L., SADEGHI, A.-R., AND WINANDY, M. 2009. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In Proceedings of STC 2009, N. Asokan, C. Nita-Rotaru, and J.-P. Seifert, Eds. ACM Press, 49–54.

DAVI, L., SADEGHI, A.-R., AND WINANDY, M. 2011. ROPdefender: A detection tool to defend against return oriented programming attacks. In Proceedings of AsiaCCS 2011, R. Sandhu and D. Wong, Eds. ACM Press.

ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. 2006. XFI: Software guards for system address spaces. In Proceedings of OSDI 2006, B. Bershad and J. Mogul, Eds. USENIX, 75–88.

FRANCILLON, A., PERITO, D., AND CASTELLUCCIA, C. 2009. Defending embedded systems against control flow attacks. In Proceedings of SecuCode 2009, S. Lachmund and C. Schaefer, Eds. ACM Press, 19–26.

HEX-RAYS. 2012. http://www.hex-rays.com/idapro/

LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. 2010. Defeating return-oriented rootkits with “return-less” kernels. In Proceedings of EuroSys 2010, G. Muller, Ed. ACM Press, 195–208.

PIN. 2012. http://www.pintool.org/