Runtime Prevention of Return-Oriented Programming Attacks

Ivan Fratric
ivan.fratric@gmail.com

Abstract

Return-oriented programming is a popular memory vulnerability exploitation technique that can be used to bypass the existing mitigation techniques such as the data execution prevention. This document describes a system called ROPGuard that can detect and prevent the currently used forms of ROP attacks. The system can be applied at runtime to any process and has a very low computing and memory overhead.

1. INTRODUCTION

Memory corruption vulnerabilities (such as buffer overflows on stack or heap, use-after-free, double-free vulnerabilities etc.) are very common in the software today [1]. In many cases, exploiting such a vulnerability can lead to the execution of arbitrary code specified by the attacker. This is accomplished by using the vulnerability to redirect the execution flow to the malicious code. Current mitigation techniques such as DEP and ASLR attempt to prevent the exploitation of these vulnerabilities, but can be circumvented in many cases.
In particular, Data Execution Prevention (DEP) is an attempt to prevent the exploitation of these attacks by adding a special flag (NX bit) to memory pages to indicate whether they contain executable code or not. Attempting to transfer execution flow to the non-executable memory page will result in an exception. To circumvent this protection mechanism, a technique called Return-Oriented Programming (ROP) has emerged [2]. The main idea of ROP is using the existing executable code pieces (in the context of ROP often called *gadgets*) to perform actions intended by the attacker. By stitching multiple gadgets together, complex behavior can be achieved.
The use of ROP is somewhat mitigated with Address Space Layout Randomization (ASLR). Among other things, ASLR randomizes the base address of executable modules that support it. This means that the attacker will (in general) no longer know the address of gadgets needed to perform the ROP attack. However, ASLR can be circumvented by making the application load a non-ASLR module, using a secondary vulnerability to perform memory disclosure or even by using the same vulnerability to perform both memory disclosure and code execution. Thus, in practice, ROP can still be used to exploit many memory corruption vulnerabilities despite the existing protection mechanisms.
This paper presents a system called ROPGuard that can detect and prevent currently used forms of the ROP attack. The system can be applied at runtime to any process and has a very low computation and memory overhead.
The rest of this document is organized as follows: In Section 2, a short introduction to return-oriented programming is given for the readers not already familiar with this technique. Related work on the existing systems for the return-oriented programming prevention is given in Section 3. The proposed system is described in Section 4. Section 5 gives the instructions on running the system and provides some implementation and configuration details. In Section 6, the results obtained through experimental evaluation of the system are presented. A brief conclusion is given in Section 7.

2. A SHORT INTRODUCTION TO RETURN-ORIENTED PROGRAMMING.

The aim of this section is to provide a short introduction to return-oriented programming to a reader not already familiar with it. If you are already familiar with return-oriented programming, you should skip this section.
The main idea of ROP is to use the existing executable code bits (gadgets) to perform actions specified by the attacker. In general, a gadget needs to be composed of two parts:
- A part that performs some action useful to the attacker.
- A part that transfers the execution flow to the next gadget.

In the most basic form of ROP attack, it is assumed that the attacker has the control over the stack. In this case, the RETN instruction can be used to transfer the execution to the next gadget. Thus, such RETN-based gadget is a sequence of instructions ending with a RETN instruction. A gadget performs some action and the attacker makes sure that, when the RETN instruction is reached, the address of the next gadget is on the top of the stack. Thus, when RETN gets executed, the program flow will be transferred to the next gadget. An example of such a ROP gadget would be

```
POP EAX; RETN;
```

Assuming that the attacker has control over the stack, the first instruction would pop an arbitrary attacker-supplied value off the stack into the EAX register while the second instruction would pop an attacker-supplied address off the stack (possibly the address of the next gadget) and redirect the execution flow into it.
Note that gadgets do not necessarily have to be small sequences of instructions as ROP can be used to stitch the existing functions together. In such a chain, the attacker transfers the execution flow to the beginning of an existing function and aligns the stack in such a way that the return address of this function is in fact the address of the next function that the attacker wants to execute (or, in general, the address of the next ROP gadget). Also note that the RETN instruction is only one way to transfer the execution flow from one gadget to the other. Indirect jump (for example, JMP [EAX]) and indirect call instructions can be used for this purpose as well [3]. This versatility makes detecting ROP exploitation attempts difficult. However, RETN-based gadgets are still most commonly used because they are most commonly found in the existing executable module.
The most common use of ROP chains in exploit writing is to make an attacker-supplied code executable and transfer code execution into it. A real-world example of such ROP chain whose purpose is to execute the attacker-supplied code located on the stack is given in Listing 1 [4]:

```
0x7c37653d, # POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
0xffffdff, # 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]
0xffffffc0, # 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]
```

Listing 1: An example of ROP chain for executing the code on the stack

This ROP chain uses the code inside msvcr71.dll to execute the potentially malicious code located on the stack. To begin executing it, the attacker must gain control of the stack (for example, through buffer overflow on stack) and align it so that the first address (`0x7c37653d`) is returned into (for example, align it with the return address of the function containing the buffer in a buffer overflow scenario). The chain works like this:
First, when the RETN is executed, execution is transferred into "`POP EAX; POP EDI; POP ESI; POP EBX; POP EBP; RETN`". This means that EAX becomes `0xffffdff`, EDI becomes `0x7c347f98`, ESI becomes `0x7c3415a2`, EBX becomes `0xffffffff` and EBP becomes `0x7c376402`. When this is executed, the address `0x7c351e05` is at the top of the stack, so it will be the address of the next gadget.
Thus, execution is transferred into "`NEG EAX; RETN`", which is the second gadget. After the first instruction is executed EAX becomes `0x00000201` and `0x7c345255` is on the top of the stack. When RETN is executed, execution flow in transferred into "`INC EBX; FPATAN; RETN`" which makes EBX zero and transfers code execution into the next gadget etc.
Eventually, the function `VirtualProtect()` is called with stack aligned in such a way that the first argument of `VirtualProtect()` is an address on the stack. This makes the stack executable and in the next gadget (`PUSH ESP; RET`), the execution flow is transferred to the stack, where the code supplied by the attacker is waiting to be executed.

3. RELATED WORK

Several projects have already been published related to the prevention of return-oriented programming. Davi et al. [5] noted that most ROP gadgets are small. They use the PIN instrumentation framework [6] to count the number of instructions between two RETN instructions. If this number is sufficiently small for several consecutive RETNs, this will be reported as an exploit attempt. The drawback of this approach is that gadgets are not necessarily small (an entire function can be used as a gadget) and chains of instructions with small

number of instructions between two RETNs can be found in the valid applications. Thus, this approach could have many missed detections and false positives in practice. Also, runtime instrumentation adds a large overhead. In the later work [7], instead of counting instructions between two RETN instructions, the authors use a shadow stack approach. This means that every time a CALL instruction is executed, the return address will be placed in a special structure called the shadow stack along with on the "real" stack. When a RETN instruction is executed, a check will be made if the addresses on the top of the real stack and on the top of the shadow stack are the same. If there is a mismatch, this will be considered a ROP attempt. The drawback of this approach is that it protects only against the gadgets that use RETN to connect with other gadgets. Also, the authors report that the runtime instrumentation introduces a runtime overhead of 2x. A similar system is presented in [8] but this system aims to also prevent gadgets that use indirect jump/call instructions to transfer execution between gadgets. This system allows indirect jumps only within the code of the same function. Additionally, on each indirect call a check is made to ensure that the target address is the prologue of some function.

Onarlioglu et al. [9] presented a comprehensive protection against ROP at the compiler level. When a program is compiled, the machine code is changed in such a way that all unintended ROP gadgets are removed (for example, RETN opcodes found as a parts of other instructions). Additionally, in its header, every function scrambles its return address by xoring it with a random key. Before the return is executed, the return address is restored. Thus, if execution flow is ever transferred in the middle of a function without first executing the function prologue, when the RETN is reached, execution will be transferred to an unpredictable location which will most likely result in an exception. The authors also added cookies on the stack of every function that has indirect jumps/calls. Before executing the jump/call the cookie is checked. In this way, this system protects from both RETN-based and the jump/call-based gadgets. The authors report an average overhead of 1.09%. Despite the comprehensive approach, it suffers from the inherent flaws of compiler-based solutions; firstly, the source code of the application needs to be available in order to apply the protection to it and secondly, in order for the protection to be effective, it needs to be applied to all executable modules in the process. Although, in theory, such protections could be applied at runtime via static binary rewriting, the rewriting process is often error prone due to problems like the need to recompute indirect jump/call targets etc.

Although it isn't aimed specifically for the protection of ROP attacks, in an impressive work, Bania [10] used static binary rewriting to add protection to the Windows kernel modules. To mitigate possible errors caused by static binary rewriting, the author left all of the original code at the original offsets and added the rewritten code after the original code. This means that every indirect jump/call whose target was not recomputed would end at the correct place in the original code. However, that also means that the rewritten modules will be twice the size of the original modules.

4. PREVENTING RETURN-ORIENTED PROGRAMMING AT RUNTIME

In this document, a system for preventing return-oriented programming attacks called ROPGuard is described. The system does not need to know the source code or any other information about the program it protects which ensures that it can be applied to any process and does not suffer from the common problems of compiler-level solutions. Also, it does not rewrite the entire executable code which ensures the stability of the protected process and low memory overhead. Instead, the protection can be applied to any process at runtime, even if the process is already running. As will be demonstrated in Section 6, the system is very fast and does not cause any stability issues in the protected applications.

The system is based on the key observation that, if we disregard the attacks that can be done fully within the context of the exploited process (such as, for example, XSS attacks), the attacker must use ROP code to leverage the attack and in this process interact with other processes or the OS kernel. Examples of such interaction include creating other processes, opening and writing to files etc.

Based on this observation, we can define a concept of *critical function*: A critical function is a function by executing which the attacker can leverage the attack, either by making modifications to the memory of the current process such that using ROP is no longer necessary or by "escaping" the current process. Some examples of critical functions include:

- CreateProcess – by calling which the attacker can create another process which can then be used to perform some malicious action and compromise the user's system
- VirtualProtect, VirtualAlloc, LoadLibrary – using which the attacker can make arbitrary code executable and no longer need to use ROP code in later stages of the exploit.
- OpenFile – by calling which the attacker can open and possibly write to arbitrary file (using WriteFile function, which may also be consider critical)

An important observation is, that in order to exploit the vulnerability, the attacker will need to call at least one critical function *from the ROP code*. ROPGuard uses this observation to perform the checks only when one of the critical functions gets called: when a critical function gets called, the appropriate checks will be performed to

determine if the critical function was called from the ROP code or as a part of normal program execution. At each critical function, ROPGuard attempts to answer the following four 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?

Performing checks only on critical function calls can create a very efficient protection mechanism in terms of performance overhead.

It is important to note that ROPGuard does not contain a hardcoded list of critical functions. Instead, critical functions are defined in ROPGuard's configuration file. In this way, critical functions can be added at any time to improve security and even process-specific critical functions can be added. Similarly, critical functions can be removed in order to improve the performance of the system.

It is also important to note that the current prototype protects only the functions in the user mode. To prevent the attacker from bypassing these functions, the same protections could also be added to the kernel counterparts of the defined critical functions.

In the remainder of this section, six checks are defined which can be performed at each critical function call to determine if the critical function was called from the ROP code and answer the questions defined earlier in this section. All of these checks can be enabled/disabled and configured in the ROPGuard's configuration file.

4.1. Checking the stack pointer

In the most forms of ROP attack, the attacker will need to have control over the stack, as demonstrated in Section 2. When exploiting buffer overflows on the stack, this comes as a given, however when exploiting other memory-corruption vulnerabilities the attacker needs to use other tricks to control it. Assuming that the attacker controls EIP and a single general-purpose register (for simplicity, let's assume EAX), then one of the common ways to get control of the stack is to move the value from EAX to the stack pointer. Thus the memory pointed to by EAX will become the "stack". This can be accomplished using the following ROP gadgets [2]:

```
MOV ESP, EAX                    XCHG EAX, ESP
RETN                            RETN
```

Unless EAX already points on the stack, doing this will have the consequence of stack pointer no longer pointing to the memory region designated for the stack when the thread in which the attack takes place was created. The designated area for the stack of the current thread can be obtained at runtime from the undocumented thread information block [11].

Thus, at each critical function call, ROPGuard checks if the stack pointer is inside the boundaries of the current thread's stack. If it is not, this is a strong indication of the ongoing ROP attack.

4.2. Looking for the address of critical function on the stack

If a critical function was entered via RETN to its address instead of via a CALL (or a JUMP) instruction, then, at the moment of entering the critical function, the address of the critical function must be on the stack just above the current stack pointer. It can be either immediately above the stack pointer if RETN instruction (opcode C3) was used to enter the critical function or (n+1)*4 bytes above the stack pointer if RETN n instruction (opcode C2) was used to enter the critical function).

Following this observation, upon entering the critical function, ROPGuard saves a certain number (specified in the configuration file) of DWORDS on the stack from being modified and later checks if the address of the current critical function can be found there. If it can, then it is possible that the critical function got entered via RETN, which could indicate that a ROP attack is taking place.

In the default configuration, ROPGuard saves only 4 bytes (single DWORD) on the stack (which corresponds to using RETN instruction without an argument to enter critical function) in order to avoid the false positives. This value can be easily modified in the configuration file.

4.3. Checking the return address

At the entry point of each critical function, its return address can be easily acquired – it is on top of the stack. Each return address must satisfy the following conditions:

- It must be executable
- The instruction at the return address must be preceded with a CALL instruction

In addition to the two checks above, ROPGuard can also verify that the target of the CALL instruction preceding the return address is the same as the address of the current critical function. This works for both direct calls and indirect calls because ROPGuard saves the state of all general-purpose registers at the point of entry of each

critical function. For example, if the instruction preceding the instruction at the return address was CALL [EAX], then ROPGuard can check if the address contained in the EAX points to the address of the current critical function. Note that, if EAX points to unreadable memory location, ROPGuard will cause an exception at this point rather than displaying an alert, but in any case, the exploitation attempt will be stopped (if this protection is enabled).
Note that, in practice, there are some cases in which the call target will not be the same as the address of critical function. One example of this are calls whose target is a jump instruction that redirects to the critical function. ROPGuard can detect most of these cases but, in order to avoid the possibility of false positives, call target checking is disabled in the default configuration (while the two checks described in the beginning of this subsection are enabled).

4.4. Checking the stack frames

If the program is compiled in such a way that it uses EBP register as a stack frame pointer, this information can be used to check the consistency of the information on the stack and check not only the return address of the critical function (as described in the previous subsection), but also to check the return addresses of the functions the current critical function was called from if they also use the frame pointer.
For example, if the program is compiled to use EBP only as the stack frame pointer, and at any frame EBP pointing to the location outside the stack is encountered, this is a strong indication that the stack is corrupted and that the attack is in progress.
The algorithm for performing the stack frame checks is given in Listing 2.
Since, in general, we don't know how the program is compiled this protection is disabled in the default configuration and is intended to be used with applications compiled so that they would not omit the frame pointers. During the experiments it was observed that most Windows components use EBP register only as a stack pointer (although some functions don't use frame pointer at all) and the false positives, when this protection was enabled were caused by the third-party applications and plug-ins (for example Flash, Skype, PGP during experiments on Internet Explorer).
In most cases, false positives caused by this can be avoided by setting the value of "RequireFramePointers" configuration option to false. When this option is set to false, ROPGuard will assume that some of the functions use EBP for other purposes than as a frame pointer, so whenever the value of EBP is encountered that does not point to the stack, the check procedure returns normally rather than alerting the user about the ROP attempt. However, in some applications, it is still possible to encounter the cases where EBP points to the stack, but is not used as a frame pointer. In the future work, this might be resolved by making a whitelist of modules that are known not to omit the frame pointers.

```
If the current frame pointer does not point on the stack
    If the RequireFramePointers option is true
          Alert the user about the ROP attempt;
    Else
          Return;
If the frame pointer points above the stack pointer
      Alert the user about the ROP attempt;
For the number of frames specified in the configuration file
    Get the return address of the current frame; //below the frame ptr.
    If the return address is null
          Return;          //reached the bottom of the stack
    If the return address is not executable or not preceded by call
          Alert the user about the ROP attempt;
    Acquire the new frame pointer //pointed to by the current frame ptr.
    If the new frame pointer is not below the previous frame pointer
    or does not point on the stack
        If the RequireFramePointers option is true
              Alert the user about the ROP attempt;
        Else
              Return;
```

Listing 2: Pseudocode of the function that performs the stack frames checking

4.5. Simulating the execution flow

While using the stack frames to check the return addresses of the functions that the critical function will return into can be very useful in detecting ROP, it has a drawback that the programs must be compiled to use frame pointers. Instead, by observing that the most ROP gadgets are short sequences of instructions ending in, most

often, RETN, we can attempt to simulate the execution of a small number of instructions after the return from the critical function. In this way, we can examine what will happen after the critical function gets called without relying on the frame pointers.
In the current version, ROPGuard simulates the instructions that change the stack pointer (PUSH, POP, ADD ESP, #, SUB ESP, #, RETN) and tracks the changes that these instructions will make to the value of the stack pointer. The remaining instructions are simply stepped over. Once RETN instruction is encountered, ROPGuard will check if the return address is executable and if the instruction at the return address is preceded by the call instruction. In this way, we can check not only the return address of the current critical function, but also the subsequent return addresses if the current critical function got called from a RETN-ending ROP gadget. Simulation stops when either the number of instructions specified in the configuration file has been simulated, or when an instruction that changes the execution flow is (other than RETN) is encountered.
In the future work, this approach could be extended to simulate other instructions as well (including indirect CALL and JMP instructions) and track changes to other registers besides ESP, which would enable ROPGuard to detect non-RETN-ending gadgets more effectively. The algorithm used in the current ROPGuard prototype is given in Listing 3.

```
instruction_pointer = return address of the current critical functions;
For the number of instructions specified in the configuration file
    current_instruction = instruction pointed by the instruction_pointer;
    Decode current_instruction and update instruction_pointer;
    If the current instruction modifies the stack pointer
          Calculate the new value of the stack pointer;
    Else if the current instruction is RETN or RETN n
        return_address = value on top of the stack;
        If the return address is not executable or not preceded by call
              Alert the user about the ROP attempt;
        instruction_pointer = return_address
        stack_pointer = stack_pointer + (n+1)*4;
    Else if the current instruction changes execution flow
        //in the current prototype, we are only concentrating on
        //hunting the RETN-ending gadgets
        Stop the simulation;
```

Listing 3: Pseudocode for program flow simulation used in ROPGuard

As an example of this simulation algorithm and its usefulness, let's assume that the critical function got called from the following ROP gadget

```
CALL [EAX]; NEG EBX; RETN;
```

and that stack is aligned so that RETN in this gadget will direct program flow to a non-executable area of memory (for example, the current critical function is VirtualProtect and the attacker's intention is to use it to make some memory page executable and then return into this memory page). In this case, in the critical function, the return address will point to the NEG EBX instruction. Also, in this case, the critical function's return address check as described in Section 4.3 will pass. However, in the simulation check (see Listing 3), ROPGuard will decode the instruction at the return address (`NEG EBX`), step over it and take the next instruction which is RETN. On encountering the RETN instruction, ROPGuard will check the return address. As, in this example, the return address is not executable, an attack attempt will be reported.
To enable program flow simulation after the return from the critical function, the number of DWORDs that will be taken from the stack by each critical function must be known and provided in the configuration file (most often, for Windows API functions, this number is the same as the number of parameters).

4.6. Function-specific checks

Other than the general checks described in the previous subsections, additional checks can be performed related to the arguments of the critical functions in order to check if executing the critical function could potentially harm the system. ROPGuard currently implements two function-specific checks, but additional checks could easily be added to the system.

- ROPGuard can prevent VirtaulProtect calls in which the program attempts to change the memory protection options of the stack. Calling VirtualProtect to make stack executable is one of the common ways of leveraging ROP attacks [4] and is also used in Listing 1.

- ROPGuard can prevent LoadLibrary (and similar) calls in which a program attempts to load a library over SMB. Loading a library over SMB is a well-known [2] method using which arbitrary code can be executed using a single (critical) function call.

These checks can easily be enabled/disabled in the ROPGuard's configuration file (they are enabled by default).

5. ROPGUARD USAGE AND IMPLEMENTATION DETAILS

ROPGuard consists of three files: an executable ('ropguard.exe'), a dll ('ropguarddll.dll') and a configuration file ('ropsettings.txt'). All three files must be in the same folder. ROPGuard can be used in two ways: to protect an already running process or to start a new protected process. To protect an already created process, ROPGuard should be started as

```
ropguard.exe PID
```

where PID is the process identifier number of the process that should be protected. To create a new protected process, a similar syntax is used

```
ropguard.exe "command_line"
```

where command_line is the command that will create the process. For example, to start a protected Internet Explorer process on Windows 7 64-bit, assuming ropguard.exe is in the path, the following command should be used.

```
ropguard.exe "c:\Program Files (x86)\Internet Explorer\iexplore.exe"
```

It is important to note that the current prototype of ROPGuard can only be used to protect 32-bit processes. There is nothing preventing the implementation of the same protection mechanisms for 64-bit processes as well, however, due to the instruction set specifics, this would mean that much of the ROPGuard's code would have to be written anew with the new instruction set in mind (for example, the code related to the patching of critical functions or execution flow simulation). As most processes, even on 64-bit windows, are 32-bit, or provide 32-bit versions (such as Internet Explorer), this limitation of the current prototype won't affect most applications.

When ROPGuard is invoked to protect an existing process it will inject its dll into the address space of the target process. This dll will then perform all of the work related to the protection of the process and the main ROPGuard application (ropguard.exe) will exit. ROPGuard uses CreateRemoteThread method for injecting its dll into the target process [12].

Invoking ROPGuard to create a new protected process was initially implemented so that it first creates the target process in the suspended state, then inject its dll into it, after which the execution of the target process is resumed. This method worked reliably on Windows 7 64-bit, however, it was observed that injecting a dll into a suspended process of some GUI applications on Windows XP causes them to not display correctly. That's why an alternate method was developed which patches the entry point of newly created process to be stuck in an infinite loop and only injects the dll once this loop has been reached, after which the application is "unstuck". This method has worked reliably on all test systems, but can still be reverted to the initial (simpler) one by seeing the "WaitEntryPoint" option to "false" in the configuration file.

Both when protecting an already started process and when starting a new protected process, once the dll has been injected into the target process, it first parses the configuration file and then proceeds to inline-patch all of the critical functions defined in the configuration file to perform appropriate ROP checks when called. If a ROP attempt is discovered during these checks, a message box with the details about the problem will be displayed and the user can choose to terminate the process or continue the execution. The patching process can be described as follows.

1. The header of the critical function is determined and saved. The header must be at least 5 bytes in size. Note that ROPGuard does not rely on a standard function header (MOV EDI, EDI; PUSH EBP; MOV EBP, ESP) and the function header does not have to be exactly 5 bytes long. Instead, function header is determined automatically by decoding and storing first few instructions of the function until they accumulate 5 bytes in size.
2. A patched function header is constructed in the executable memory region allocated previously by ROPGuard. A patched header consists of the instructions shown in Listing 4. As can be seen in the listing, in the patched function header, a part of stack is saved for later examination (see Section 4.2.) along with the state of all general-purpose registers, the RopCheck() function is called which performs the checks described in Section 4 and finally, the execution of critical function is resumed.
3. Function header is replaced by a jump instruction to the patched function header.

```
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;
```

Listing 4: Patched function header

One important exception to the above behavior is the CreateProcessInternalW function. In most cases, if the protected process spawns a child process, we'd like to protect the child process as well. A good example where such behavior is desirable is Internet Explorer, which spawns a new process for every browser tab/window. In case the protected process wants to spawn a child process, in the patched CreateProcessInternalW function, ROPGuard first performs appropriate checks to verify that CreateProcessInternalW was not called from the ROP code. After that, it starts the child process and injects the ROPGuard dll into it (which starts the protection of the child process), after which the execution of the child process is resumed and the CreateProcessInternalW function in the parent process returns. Note that this behavior can be turned off in the configuration file.
During profiling of ROPGuard, it was observed that the majority of time in ROPGuard was spent on the VirutalQuery function which is used to determine if a return address of a function is executable. This is why an option was added in ROPGuard to cache the VirutalQuery results. This cache requires using a process-level mutex (implemented as CRITICAL_SECTION) to protect the cache from simultaneous modification by multiple threads. However, since cache use is rare and very fast, this mutex is usually unlocked, so very little overhead is expected. Also, using the cache requires clearing the cache on any function that could potentially make executable memory non-executable (for example, VirtualProtect, VirtualFree etc.). These functions can be defined in the configuration file.
Note that, because of the way ROPGuard works (dll injection, inline dll patching), it might not be compatible with some anti-virus software.

5.1. ROPGuard configuration

ROPGuard can be configured by editing the "ropsettings.txt" text which should be in the same folder as the ROPGuard executable. The file has a very simple syntax where each line has the form

```
option = value
```

where "option" is the name of the option being set, and "value" is the value to set it to. The lines beginning with the '#' character are considered comments and are not parsed.
The configuration file can be used to

- enable/disable different checks described in Section 4 and configure their properties
- enable/disable a message box when a process is successfully patched
- define critical functions

All options are described in the default configuration file that comes with the ROPGuard prototype. The reader should see the example configuration file to see the available configuration options and their default values.
The default configuration file is designed to provide the maximum protection to any process regardless of how it was compiled without being susceptible to false positives. This is why the two checks that could possibly result in false positives are disabled in the default configuration: stack frames check (Section 4.4.) and the stack check described in Section 4.2. The stack frame check (Section 4.4.) should be enabled to provide additional security for the applications that do not omit frame pointers. During the evaluation, it was observed that the protection described in Section 4.2. does not cause any false positives in the majority of applications (including Internet Explorer 9 and all benchmark runs on Windows 7 64-bit), so it can be enabled for most applications for added security.
With the prototype, an alternate configuration file ("ropsettings-frames.txt") is also provided which is designed to provide additional protection for the applications that were compiled in such a way that they do not omit frame pointers. To use this configuration file, rename it to "ropsettings.txt" (after moving the default configuration file). Note that this configuration file defines less critical functions than the default configuration file. This is because, by using stack frames, we can find the stack frames of other possibly critical functions if they call the defined critical function (see Section 4.4.). For example, it might be possible to omit CreateProcessA from the list of critical functions because it will always call CreateProcessInternalW (which is

defined as a critical function). In this case, during the stack frame check of CreateProcessInternalW, the stack frame of CreateProcessA will also be traversed.

6. EXPERIMENTAL EVALUATION

Several experiments were performed using ROPGuard to test its stability, efficiency in detecting ROP exploitation attempts, reliability to false detections and overhead introduced by the protection.
Firstly, ROPGuard was used to test the normal running of Windows applications such as Notepad, Microsoft Office and Internet Explorer. Experiments were performed on multiple systems running Windows 7 64-bit and Windows XP 32-bit. Special attention was given to testing ROPGuard with Internet Explorer, as it is probably the most security-exposed application. It was observed that, with the default ROPGuard configuration, there were no false positives or stability issues during the evaluation.
Secondly, an example vulnerable application and the appropriate exploit were written to test the ROPGuard's efficiency in detecting the exploitation attempts. The vulnerable application (provided with the prototype as 'vulnapp.exe') reads the first line of a file given as the first argument into a buffer of fixed size (64 bytes) located on stack. If the line is larger than 64-bytes, buffer overflow will occur. For simplicity, the application was compiled without the stack protection (GS switch). A special input file that will cause the buffer overflow, execute the ROP code given in Listing 1 and after that execute the example payload ('calc.exe' will be started on successful exploitation) was crafted. When the application is protected with ROPGuard and normal input is given to the vulnerable application, the application runs normally. However, when the crafted input (available as 'vulnapp-input-rop.txt') is given to the vulnerable application and the application is protected using ROPGuard, ROPGuard will detect the exploitation attempt successfully. In particular, ROPGuard can detect the payload given in Listing 1 in several different ways:

- Firstly, during the VirtualProtect's return address check (Section 4.3.), ROPGuard will detect that the instruction preceding the one at the return address (0x7c376402) is not a call instruction.
- If stack frames check is enabled (Section 4.4.), which is possible because the vulnerable application does not omit frame pointers, two things will be noted:
    - That the return address (0x7c37653d) of a frame "below" the critical function is not preceded by a call instruction
    - That the frame pointer will eventually end up outside of the stack
- If the two protections above were skipped, during the program flow simulation (Section 4.5.), when the next RETN after the return from the critical function is reached, it will be determined that the return address on the stack (0x7c345c30) is again not a valid return address (because, again, it is not preceded with the call instruction).
- If all of the above protections were skipped, finally, during the function-specific protections check (Section 4.6.) it will be detected that VirtualProtect is being called to change the access rights of the stack.

The ability to detect this exploitation attempt in four different ways demonstrates the ROPGuard's effectiveness in detecting ROP exploitation methods currently used by the exploit writers.
To test the ROPGuard's computing overhead, a series of benchmarks was performed with and without the protection. The default configuration was used. Benchmarks were performed on a computer with an i7 720QM processor and 4GB RAM running Windows 7 64-bit. The benchmark results are given in Table 1. The results were given with and without the executable module cache (as described in Section 5).

Benchmark name	Benchmark type	Score, not protected	Protected, no cache		Protected, with cache	
			Score	Overhead	Score	Overhead
PCMark Vantage	System	5049*	5009*	0,80 %	5024*	0,50%
NovaBench	System	799*	784*	1,91%	789*	1,27%
Peacekeeper	Browser	1480*	1406*	5,26%	1481*	-0,07%
SunSpider	Browser	247,7 s	253,8 s	2,46%	248,3 s	0,24%
3DMark06	Gaming	7994*	7992*	0,03%	7996*	-0,03%
SuperPI 16M	CPU	403,0 s	399,5 s	-0,87%	406,9 s	0,97%
Average overhead				**1,59%**		**0,48%**

Table 1: ROPGuard overhead in different benchmarks. Scores marked with * are expressed in custom units used by the corresponding benchmarks and the larger score is better

Due to the fact that i7 processor used in the experiments can change its clock frequency at runtime, the results may vary for different runs. This is especially visible during the CPU-heavy tests like in the SuperPI benchmark where the best run with protection was faster than the best run without protection. To mitigate this, each

benchmark was run multiple times and only the best score was taken. However there is still some noise present in the measurements. Despite this, average scores successfully demonstrate very low computing overhead introduced by ROPGuard, especially if the caching option is enabled.
ROPGuard dll, which is loaded into any protected process, takes 48 kB in the address space of the protected process. Additional memory used by ROPguard, such as the memory used for patched function headers, configuration and the executable module cache is very small (4 kB for patched function headers and approximately 5 kB for the configuration and cache). As ROPGuard modifies Windows dll code at runtime, additional memory overhead is introduced automatically through the copy-on-write memory page protection mechanism [13]. The amount of memory overhead introduced this way depends on the number of modules for which critical functions are defined and their size. For example, if only kernel32.dll (which contains the majority of critical functions) is patched, the introduced overhead will be less than 1 MB.

7. CONCLUSION

Return-oriented programming is a popular memory vulnerability exploitation technique that can be used to bypass the existing mitigation techniques such as the data execution prevention. A system called ROPGuard was developed that can detect and prevent the currently used forms of ROP attacks. The system works by defining a set of critical functions: functions that need to be called from the ROP code by the attacker in order to leverage the attack. A series of checks is performed on each critical function call to determine if a function was called from the ROP code or as a result of normal program execution. The system can be applied at runtime to any process and has performed very reliably during the evaluation. The system added an average computing overhead of 0,48% during the evaluation and has a small memory overhead which makes it very CPU- and memory-efficient.
In the future work, the execution flow simulation, which is found to be one of the most promising directions of this work could be extended to simulate the full instruction set instead of just tracking the stack pointer changes and RETN instructions. This would make ROPGuard more suitable for detecting CALL and JMP-based ROP gadgets. One of the possible directions for future research is also combining the techniques proposed here with the compiler-level approaches (such as adding special structures that would enable more reliable reconstruction of the call stack instead of relying on frame pointers and execution flow simulation for this purpose).

8. REFERENCES

[1] National Vulnerability Database, http://nvd.nist.gov/
[2] Dino Dai Zovi. Practical Return-Oriented Programming. SOURCE Conference, Boston 2010.
[3] S. Checkoway and H. Shacham: Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical report, 2010.
[4] Corelan ROPdb, https://www.corelan.be/index.php/security/corelan-ropdb/, 2011.
[5] L. Davi, A.-R. Sadeghi and M. Winandy: Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. Proceedings of the ACM workshop on Scalable trusted computing, pp. 49-54, 2009.
[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood: Pin: Building customized program analysis tools with dynamic instrumentation. In PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190-200, 2005.
[7] L. Davi, A.-R. Sadeghi and M. Winandy: ROPdefender: a detection tool to defend against return-oriented programming attacks. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 40-51, 2011.
[8] P. Chen, X. Xing, H. Han, B. Mao and L. Xie: Efficient Detection of the Return-Oriented Programming Malicious Code, Proceedings of the 6th international conference on Information systems security, pp. 140-155, 2010.
[9] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti and E. Kird: G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries, Proceedings of the 26th Annual Computer Security Applications Conference, pp. 49-58, 2010.
[10] P. Bania: Securing The Kernel via Static Binary Rewriting and Program Shepherding, 2011.
[11] Win32 Thread Information Block, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Win32_Thread_Information_Block, accessed in January 2012.
[12] DLL injection, Wikipedia, the free encyclopedia , http://en.wikipedia.org/wiki/DLL_injection, accessed in January 2012.
[13] Memory protection, http://msdn.microsoft.com/en-us/library/windows/desktop/aa366785(v=vs.85).aspx, Microsoft, 2012.