ROPInjector: Using Return Oriented Programming for Polymorphism and Antivirus Evasion

Giorgos Poulios, Christoforos Ntantogian, Christos Xenakis

Department of Digital Systems, University of Piraeus
Piraeus, Greece
{gpoulios,dadoyan,xenakis}@unipi.gr

Abstract

The downside of current polymorphism techniques lies to the fact that they require a writeable code section, either marked as such in the corresponding Portable Executable (PE) section header, or by changing permissions during runtime. Both approaches are identified by AV software as alarming characteristics and/or behavior, since they are rarely found in benign PEs unless they are packed. In this paper we propose the use of Return-Oriented Programming (ROP) as a new way to achieve polymorphism and evade AV software. To this end, we have developed a tool named ROPInjector which, given any piece of shellcode and any non-packed 32-bit Portable Executable (PE) file, it transforms the shellcode to its ROP equivalent and patches it into (i.e. infects) the PE file. After trying various combinations of evasion techniques, the results show that ROPInjector can evade nearly and completely all antivirus software employed in the online VirusTotal service. The main outcome of this research is the developed algorithms for: a) analysis and manipulation of assembly code on the x86 instruction set, and b) the automatic chaining of gadgets by ROPInjector to form safe, and functional ROP code that is equivalent to a given shellcode.

1. Introduction

Return Oriented Programming (ROP) gained increased attention during the late 2000's [3] as an advanced stack smashing method that could bypass Data Execution Prevention (DEP) mechanisms. ROP is a rediscovery of threaded code in which programs typically consist of a chain of addresses in the stack pointing to code chunks in the attacked executable (or its loaded libraries) each of them ending with a return instruction (commonly `ret` but not only). These borrowed code chunks are called *gadgets* and their "return" is in fact a call to the next gadget in the chain. As an analogy to regular code, in ROP, gadgets are the "instructions" and esp is the program counter.

Polymorphism is a technique for AV bypass in which the code changes itself each time it runs, but the function of the code (its semantics) does not change at all. In this way, AVs cannot create a signature for detection of the shellcode. However, the downside of current polymorphism techniques lies to the fact that they require a writeable code section, either marked as such in the corresponding PE section header, or by changing permissions during runtime. Both approaches are identified by AV software as alarming characteristics and/or behavior, since they are rarely found in benign PEs unless they are packed.

In this work, we claim Return-Oriented Programming (ROP) to be a strong polymorphism alternative that eliminate the need of writable code section. More specifically, the first and most important benefit of using ROP for AV evasion is that such borrowed code (that of gadgets) is always benign and tested against false positives. Evidently, the return address chain has to be built somehow onto the stack and that would leave a footprint subject to signing. The process involves either pushing the return addresses to the stack or just copying the whole chain from another memory location (possibly some .data segment) and adjusting the stack pointer. However, we argue that: i) the code required for such operations is very common and seemingly benign, ii) can be randomized or encoded in many and trivial ways, iii) it largely depends on the attacked PE and its image base since in the worst case it is a series of push <VA_i> operations. This holds because gadget addresses change for different PEs and different image bases, hence changing the footprint and statistics of the chain building instructions even if they originate from the same source shellcode. Given these features, ROP enables polymorphism **without requiring a writeable code section in memory**. Encoding/decoding can be applied on the gadget chain in memory (i.e. in the stack and not in the code section) and/or different gadgets can be randomly chosen for the same operation hence altering the shellcode's footprint.

Based on the above observations, in this paper we present ROPInjector, a tool which, given any piece of shellcode (hereafter, also referred to as *source (shell)code*) and any non-packed executable file, it transforms the shellcode into its ROP equivalent and patches it into (i.e. infects) the PE file. ROPInjector, which is written in Win32 C, infects Win32 Portable Executables (PEs) and works for the x86 architecture. Since it is very common for AVs to detect minor deviations from the typical arrangement of the file sections and their characteristics (e.g. a second executable section with RWX permissions), besides the transformation of the code into a non-recognizable, non-recurrent form, the developed tool addresses several additional issues to achieve evasion, such as the positioning of the shellcode in the carrier executable and the way of transferring control to the shellcode. Moreover, we have performed several experiments to evaluate the effectiveness of the proposed tool by injecting shellcodes to well-known executable files including acrobat reader, firefox, Java, etc. Quantitative results show that our proposed technique, if combined with simple behavioral anti-profiling techniques may render AV detection infeasible.

2. Related Work

To the authors' best knowledge this is the first work that infects PEs with ROP-encoded payload. Nevertheless, in this section we examine two tools having the same purpose with ROPInjector, that is, to infect PE files with common (possibly encrypted) shellcode in order to bypass AV software.

The first, Shellter [1], focuses on maintaining the original structure of the PE file, by avoiding injection of the shellcode into predefined locations or changing the characteristics of the existing sections. It achieves so by overwriting existing code for which it is certain that will be given control during execution of the program. The latter is deduced by tracing the executable file and analyzing its execution flow. Shellter is also capable of reusing imports of the original PE file to change the writing permissions of the section containing the shellcode so that encrypted and self-modifying code can be used. It is also capable of injecting "junk code" before the shellcode that delays execution as a means to anti-emulation. Shellter is advanced in terms of dynamically selecting the location of the patch in the shellcode (as opposed to extending the .text section). However, while it features a patching method that introduces variability (as to where in the file is the shellcode injected), it relies on traditional polymorphism methods, that are still subject to signature generation and detection of write permissions or modifications of the .text section in memory. Moreover, our proposed approach introduces variability too, due to the transformation to ROP (which is dependent on the PE file).

PEinject [2] is mostly a method (and referenced as such) rather than a full-featured tool. It injects the shellcode in the (first sufficiently large) padding space found in the .text section (either 0xCC caves or section padding) and does not encode or modify the payload in any way, neither does it anticipate for self-modifying or encrypted payloads. Control is passed to the injected shellcode by modifying the address of entry point of the PE file's NT_HEADER.

The evasion ratios of both methods are compared with our proposed approach in Section 4.

3. ROPInjector

3.1 PE patching and passing control to the shellcode

First of all, the patching of the PE file and the passing of control to the shellcode must be done in the least noticeable way. A second executable section hosting the shellcode would be too alarming, since the vast majority of executables has only one. The next least disruptive and easy to implement option would be to inject the shellcode in the 0xCC padding commonly left by the linker in-between code segments (typically OBJ files) in the .text section of PEs. However, there may not always be sufficient space in those 0xCC caves, while it is important to notice that ROPInjector puts this padding space into better use for our purposes as we analyze below.

For the above reasons we choose to append the shellcode to the existing .text section of the executable, and correct all section headers and relocations accordingly. To pass control to it, the default practice is to replace the instructions pointed to by `NT_HEADER.AddressOfEntryPoint` with a jump to the shellcode which is appended those replaced instructions followed by a jump back to the original execution flow. Directly pointing the address of entry point to the shellcode in this case is avoided, since many AVs' heuristics are alarmed by the fact that it points towards the end of .text. An alternative to giving control to the shellcode at program entry, is to hook any calls to `ExitProcess`, `exit` or other similar functions. This technique in particular, as shown also later by the results, bypasses behavioral profiling by AVs that employ emulation or sandboxing. This can be attributed to the fact that either AVs emulate only a small portion of the executable's entry code due to scanning time constraints, or because of lack of (universal) techniques for triggering a graceful exit (many programs do not handle SIGINT and SIGTERM signals).

3.2 Reverse analysis of x86 machine code

Reverse analysis of machine code into data structures that are easy to handle is crucial to perform any kind of patching, modifications, re-assembly, and any transformation to ROP.

Two are the most important pieces of information required: i) the origin and destination of all relative references (e.g. a relative jump and its target) and ii) which registers are being written or read during each instruction, as well as which registers are free to modify. The former is required for injecting or removing instructions from a code segment without breaking its functioning. The latter is particularly useful to enhance gadget matching, either by performing permutations, or by using gadgets that contain redundant but safe instructions (in this case, *unsafe* are branch, privileged, or indirect addressing mode instructions because they risk raising errors such as access violation).

3.3 MOD/REG/RM and SIB unrolling

Instructions using the MOD/REG/RM indirect addressing mode with displacement or the Scaled Index Byte (SIB) addressing scheme in the shellcode are treated specially before the transformation to ROP. Such instructions are unwanted for the following reasons:

i) They are long (in the best and not so likely case 3 bytes long: 1 for opcode, 1 for MOD/REG/RM and 1 for SIB) hence unlikely to be found in gadgets;
ii) They often read many general purpose registers at once, thus reserving them while as mentioned earlier, the more the free registers the better;
iii) Their respective gadgets (should they be found or injected) will probably not be reusable, due to the use of displacement and index constants (e.g. `mov edx, [esi*2+16]`).

In order to circumvent this kind of situations, we reduce such instructions to their arithmetic equivalents one-by-one. We call this process *unrolling* and it is performed to the shellcode before any transformation to ROP. For instance, `[1] mov eax, [ebx+ecx*2]` may be replaced by:

```
[1']     mov eax, ecx
[2']     sal eax, 1
[3']     add eax, ebx
[4']     mov eax, [eax]
```

If the register eax is not free to use for the arithmetic operations, another temporary register that is free may be used.

Noteworthy is how unrolling unlocks register access from one atomic instruction to many. For instance, in the latter example, ecx is freed at [1'] and ebx at [3']. If for example eax were to be freed at the preceding 10 instructions, then instructions [1'] to [3'] could be moved 10 instructions behind, thus resulting in an additional free register (i.e., ecx and ebx, but not eax which will not be free) in that preceding code chunk.

3.4 Finding gadgets

Candidate gadgets in the executable sections of the given PE file must end in one of `ret`, `retn`, `pop regX; jmp regX`, or `jmp regX`. Exceptionally for the latter, the gadget in question must be first paired with a *loader gadget* that loads the required return address into `regX`. The process begins by finding all gadget endings and temporarily storing them to a list. For each of those endings, *n* bytes of preceding machine code is disassembled for each *n* up to maximum depth *N* (typically 20 bytes). If such disassembly aligns with the ending (not guaranteed since x86 instructions are of variable length) a candidate gadget has been found. Candidate gadgets containing any illegal, privileged (e.g. `sysenter`, `int`, `iret`), branch or esp modifying instructions are filtered out.

3.5 Parsing gadgets into Intermediate Representation (IR)

The gadgets found in the aforementioned process are first analyzed instruction-by-instruction to infer register access. Since gadgets are allowed to contain safe but redundant instructions, their register access is tested for modifications to the register in question (e.g. a `mov ecx, eax; pop ecx; ret;` gadget cannot be used for moving eax to ecx) as well as the non-free registers of the source instruction to be encoded.

Following that, they are parsed into an Intermediate Representation (IR) consisting of an operation-type, and 3 operands with different meaning depending on the type. If a multi-instruction gadget contains more than one representable instructions, only the first is considered. However, the following ones have also been considered in other gadgets with the same ending, because of the backwards gadget finding process described in the previous paragraph. Noteworthy is the fact that by parsing into this higher level IR, one-to-one permutations are automatically performed. That is because both gadgets and instructions are classified into one of these types, based on which the encoding is then performed, rather than on the instructions per se. The IR is also useful for selecting the encoder function accompanying every gadget. Encoders are responsible to answer "*whether their assigned gadget can encode a given instruction*", as well as to encode it into a list of stack operations if requested to.

3.6 Injecting gadgets

In order to enhance transformation of the source shellcode, and since not all required gadgets are always found in the PE file, new ones are also injected as needed. Firstly, the 0xCC caves are used for this injection, and if they are filled, the .text section is extended before the actual patch. The injection is performed in the least noticeable way to avoid alarms. If a standard epilogue (`mov esp, ebp; pop ebp; ret`) is found right before the 0xCC cave, the gadget is injected in-between the preceding code and the epilogue. Figure 1 depicts such an example gadget injection of a `mov ecx, eax` gadget.

`mov esp, ebp`
`pop ebp`
`ret(n)`
`CCCCCCCCCCCCCCCCCC`
	`jmp epilogue`; *normal flow avoiding gadget*
	`mov ecx, eax`; *the injected gadget*
	`jmp return` ; *gadget flow avoiding std. epilogue*
	epilogue:
	`mov esp, ebp`
	`pop ebp`
	return:
	`ret(n)`
	`CCCCCCCC`

Figure 1: Injection of gadget (*right*) in 0xCC cave preceded by standard epilogue (*left*)

In the case that no epilogue is found at the boundary with the 0xCC cave, a pseudo-function with standard prologue and epilogue is injected to avoid heuristics or n-grams that might raise suspicion due to non-ordinary returns. This pseudo-function has the following form:

```
        push ebp
        mov ebp, esp
        <gadget code>
        jmp return
        mov esp, ebp
        pop ebp
return:
        ret
```

Figure 2: Pseudo-function ending used during gadget injection

Following gadget insertions will then reuse this pseudo-epilogue as stated above, by injecting before the standard epilogue, thus making it look more like a real function.

3.7 Source code permutations

Predefined, one-to-one permutations (i.e. one instruction to one gadget) are achieved through the IR and encoder functions. Encoders will also perform basic algebraic permutations based on the properties of addition, subtraction multiplication and division. For instance, if the instruction to be encoded is of type ADD_IMM (`add reg, imm`), an encoder will repeat anything `add reg, x` with `x` being an integer divisor of `imm`, `imm/x` times. Addition and subtraction with constants will also be swapped if the signs of the constants are flipped. M-to-N permutations quickly scale to exponentially growing space and are out of the scope of this work.

3.8 Chaining gadgets

The return address chain can be built either during runtime or during compile-time and saved to the initialized data section of the file (to be then copied at runtime to the stack). The most alarming option would be the first (during runtime) and we choose this to evaluate our evasion ratio (also chosen as an implementation option). During this process, besides the pushing of the VAs onto the stack, the ROP compiler must consider pushing immediate constants, adjustments for stack pointer modifications in the gadget (e.g. redundant `pop`s, `retn`s) and gadgets with loader gadgets. For this purpose, the following types of *stack operations* are defined:

PUSH_VA	; *push a (loader) gadget VA onto the stack*
PUSH_IMM	; *push an immediate constant onto the stack*
ADVANCE	; *advance (subtract from) the stack pointer a number of bytes*
CHAIN	; *pseudo operation denoting a placeholder for the next gadget's VA*

The result of the encoding process of a given instruction by a given gadget is a series of stack operations for the invocation of the gadget. The list of such operations for all gadget calls describes the assembly instructions that if executed, will build the chain in the stack. Alternatively, such operations may be used to create the required stack frame during compile-time, save it as initialized data and copy it over from the data section during runtime. The latter process allows also for encoding/decoding of the stack frame. In the former case, and when multiple calls are made to the same gadget (e.g. as in using `inc eax` to achieve `add eax, X`) the compiler wraps the call with a conditional jump loop using a free register.

However, not all types of instructions can be easily encoded into ROP. In this work we do not consider the encoding of branches (jumps, calls, loops, interrupts), privileged instructions and pops. Hence, the return-oriented code chunks must finally return back to the source shellcode. This is achieved by wrapping the chain building instructions in the following:

```
   [1]      call build_chain
   [2]      jmp past_the_chain
build_chain:
   [3]      push <VA of gadget N>
   [4]      ....
   [5]      push <VA of gadget 1>
   [6]      ret
past_the_chain:
   [7]      <other instructions / chains>
```

In this way, the last gadget (N) will return to instruction [2] jumping past the chain building instructions and continuing normal execution flow.

4. Experiments and Results

In order to evaluate ROPInjector we used the VirusTotal online antivirus scanning service [4] which at the time of this writing includes 57 AVs. For carrier PEs (i.e., the infected ones), we selected 9 popular 32-bit executable of various sizes that most of them also include certificates (see Table 1).

Table 1: List of PE files used as carriers in the experiments

Executable	Size (KB)	Version	SHA256 hash
AcroRd32.exe	1489	Version 10.1.12 of Adobe Acrobat Reader X	a03297789b5a784af3765c523b33b9d54578e38a178ca67103b5e0e74f905331
Acrobat.exe	321	Version 10.0.0.396 of Adobe Acrobat X Pro	281529dbd6c45cc1706d5cd66456b5c983aa5e6e3dc64723779d9b2bd48b769d
cmd.exe	296	Version 6.1.7601.17514 Windows Command Processor	17f746d82695fa9b35493b41859d39d786d32b23a9d2e00f4011dec7a02402ae
Rainmeter.exe	39	Version 2.4.0.1678 of Rainmeter	00c8f2b58ffb318cf1031f58f4fe86a73bcb9716c7072012114bd42f157dd071
firefox.exe	331	Version 35.0.0.5486 of Mozilla Firefox	11740f07a822637874da4eb4eafa309d145a1ca729779e30cb3d1e592c5484df
java.exe	172	Version 7.0.710.14 of Oracle Java	06889c037faab8379aaafb2bf9e77807e3d432da435cdab1244bff36c5c562d5
wmplayer.exe	163	Version 12.0.9600.17415 of Microsoft Windows Media Player	c7adbfeeb7993928cb542751625dac6b10e96b18fcdcd836a72cad62ae797250
nam.exe	1829	Version 1.0a11a of “The Network Animator”	5d329bb39ba744cdba5e1afe107551c18ba0acd46cb6764391024a73aa2d583f
notepad++.exe	2348	Version 6.6.9.0 of the GNU text editor for Windows	a11077cb6c209c67eb2d507d650fbee0925f3cbe860c70e0cd779b73f5af4b80

Regarding the source shellcode, we selected the two most popular payloads of Metasploit [5]: i) Reverse TCP Shell, and ii) Reverse TCP Meterpreter. For each PE and each shellcode we performed 4 patching scenarios as listed in Table 2, resulting in a total of 72 samples.

Table 2: List of patching scenarios tested against VirusTotal

Patching Scenario	**Description**
Original	The executable file is not patched at all
ROP-Exit	This is the executable file generated by the ROPInjector. The executable file is patched with the shellcode unrolled, converted to ROP, and entry point before the original program's exit (hook ExitProcess or exit)
Exit	In this scenario, the executable file is patched with the shellcode intact and entry point before the original program's exit (hook ExitProcess or exit)
Shellcode	The executable file is patched with the shellcode intact, and entry point before the original program.

Figure 3 and Figure 4 depict the evasion ratios ($1 - \frac{detection}{total\ \#\ of\ AVs}$) of ROPInjector for each one of the four scenarios for the reverse shell and the reverse meterpreter payloads respectively. We can observe that the executables generated by the ROPInjector (i.e., "ROP-Exit" scenario) achieve the highest evasion ratio. In particular, in more than half of the test cases the ROPInjector results in 100% AV evasion, while in some PE files (e.g., java.exe), the ROPInjector has evasion ratio greater than 98.5% for both the reverse shell and meterpreter shellcodes. This means that in average ROPInjector achieves AV evasion equal to **99.31%,** as depicted in Figure 5 (i.e., "ROP-Exit" scenario).

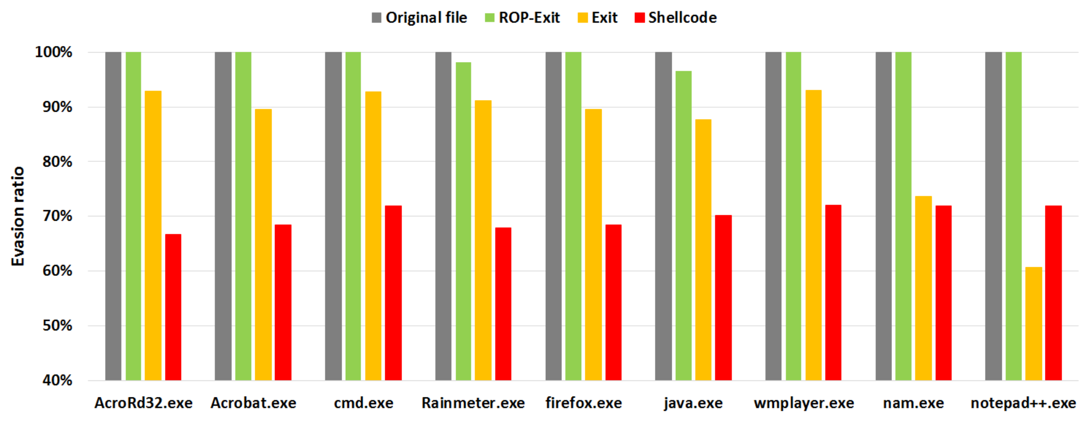

Figure 3: Evasion ratio of ROPInjector for the reverse shell payload

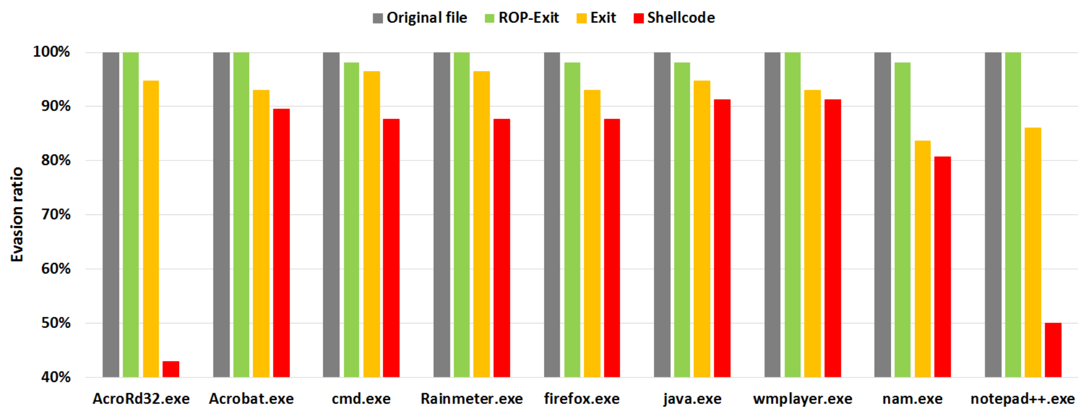

Figure 4: Evasion ratio for the reverse meterpreter payload

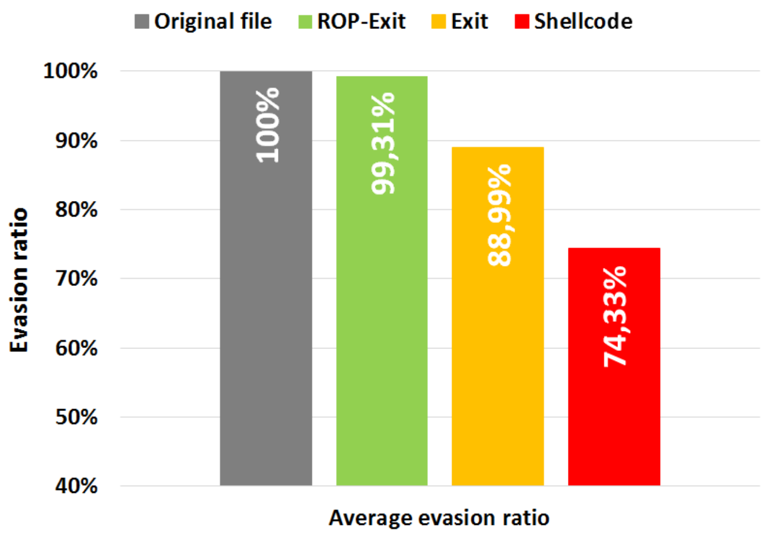

Figure 5: Average evasion ratio per combination of methods

From these results we can deduce that evasion depends almost equally on both code obfuscation/transformation (hence signature evasion) and entry point (hence behavioral profiling evasion). This can be attributed to the fact that some AVs were able to detect ROPInjector despite the fact that there is no signature, due to the ROP polymorphism. It seems that behavioral analysis is equally important to static signatures for some AVs (from the ones that were alarmed) and is mostly performed during entry of executables.

Moreover, a comparison is also made with Shellter v2.2 [1] and PEinject [2] in Figure 6. Shellter was used with its default options (i.e. with polymorphic junk code). We can observe that executables generated from our proposed ROPInjector (i.e., "ROP-Exit") have the highest evasion ratio in all conducted experiments compared to Shellter and PEinject. Note also that even the simple "Exit" scenario achieved in some executable files better results than Shellter. Finally, the peinject had the worst evasion ratio.

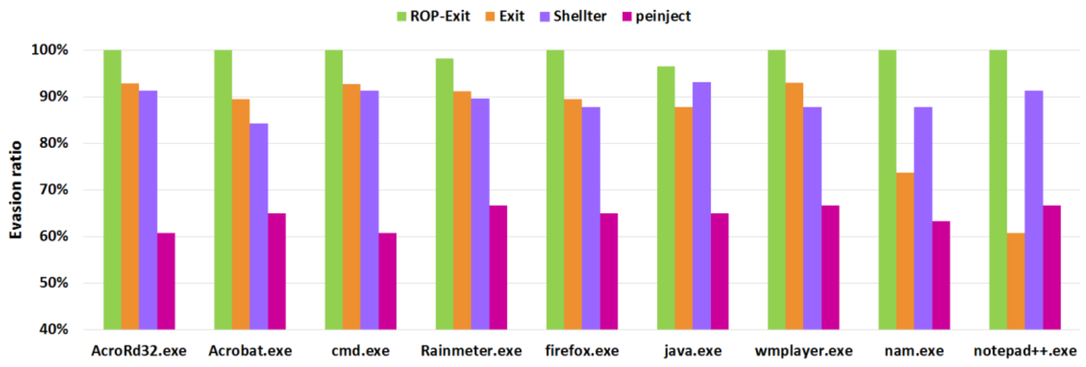

Figure 6: Comparison of evasion ratio between "ROP-Exit", "Exit" scenarios with Shellter and PEinject

It is also important to notice that besides VirusTotal, we have also tested the effectiveness of ROPInjector against a special piece of software named NCCGroup's "Experimental Windows .text section Patch Detector" [7]. This detector compares the executable sections in memory against the ones on disk to detect modifications/patching. As expected, no executable was detected as patched, since ROPInjector does not alter the .text section in memory (neither does it require to).

5. Conclusions

Most antivirus software rely on string signatures and mild behavioral profiling detection mechanisms. By encoding the shellcode into its return-oriented equivalent and even by performing elementary mutations (unrolling), the former can be bypassed in the vast majority of cases. Behavioral profiling can also be avoided by carefully intercepting normal execution flow in points that AVs either cannot emulate or simply cannot derive enough evidence to classify the behavior as malicious. In this work, we presented as a means to the latter the hooking of common calls to process exit resulting in many cases in absolute evasion and in others rates greater than 98%.

The techniques presented can still be mitigated if dealt with individually. For instance, signatures could be created for ROP building instructions (although not expected to be very effective) and behavioral analysis could be also performed backwards in terms of process life-cycle. However, since slight variations and randomization can again disarm scanners, a more robust countermeasure does not seem straight-forward to design, practical to implement, or even realistic to propose. Perhaps the most promising direction is towards the strict coupling of the host operating system with the trusted software certificates (or checksums) and a “default distrust all” policy, i.e., whitelisting rather than blacklisting, pretty much like what was started by Microsoft back in 2001 [6] but did not flourish.

References

[1] Shellter, https://www.shellterproject.com, last accessed on February 15, 2015

[2] Injecting Shellcode into a Portable Executable(PE) using Python, http://www.debasish.in/2013/06/injecting-shellcode-into-portable.html, last accessed on February 15, 2015

[3] Shacham, Hovav. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[4] VirusTotal, https://www.virustotal.com, last accessed on February 15, 2015

[5] Metasploit, http://www.metasploit.com/, last accessed on February 15, 2015

[6] Default Deny All Applications, http://www.windowsecurity.com/articles-tutorials/authentication_and_encryption/Default-Deny-All-Applications-Part1.html, last accessed on February 15, 2015

[7] Experimental Windows .text section Patch Detector, https://github.com/nccgroup/WindowsPatchDetector, last accessed on February 15, 2015