black hat USA 2015

ROPInjector: Using Return-Oriented Programming for Polymorphism and AV Evasion

G. Poulios, C. Ntantogian, C. Xenakis
{gpoulios, dadoyan, xenakis}@unipi.gr

UBM Tech

What is Return Oriented Programming

- **ROP** is an **exploitation technique** that allows an attacker to execute:

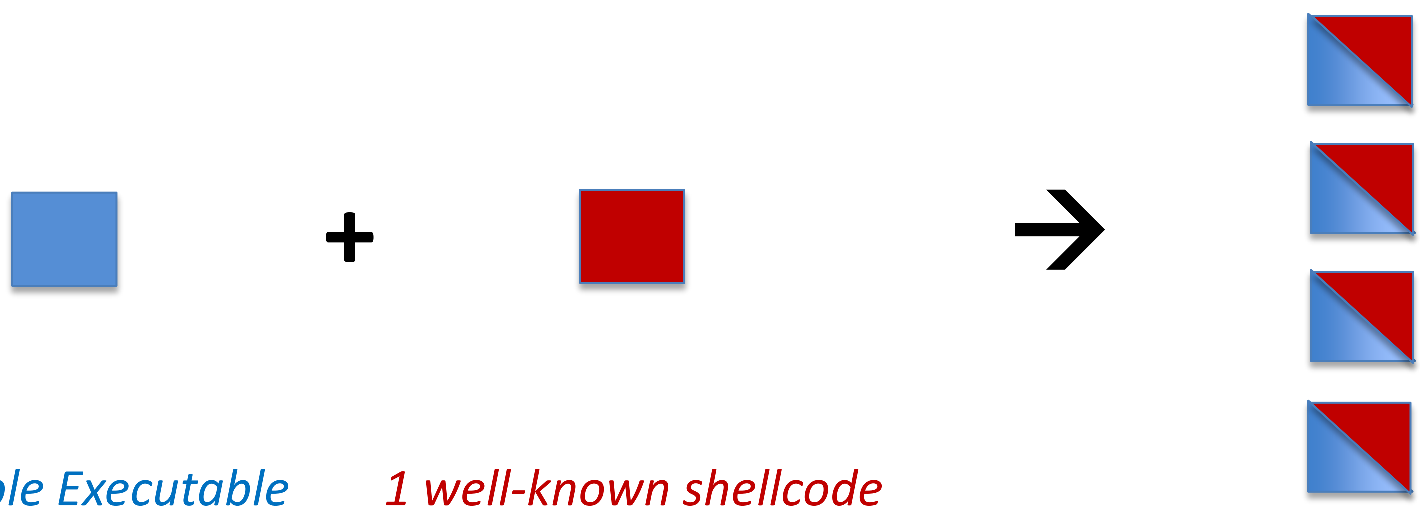
  A **sequence of machine instructions** named **"gadgets"**

- Each **gadget** is a part of **borrowed code** that **ends** with the instruction **return**

- **A sequence of gadgets** allows **an attacker** to perform **arbitrary operations**

Objective of this research

- **ROP** has been mainly used to **bypass** the **non-executable memory defense mechanism**.
- We propose **ROP** as a **polymorphic alternative** to achieve **AntiVirus (AV) evasion.**

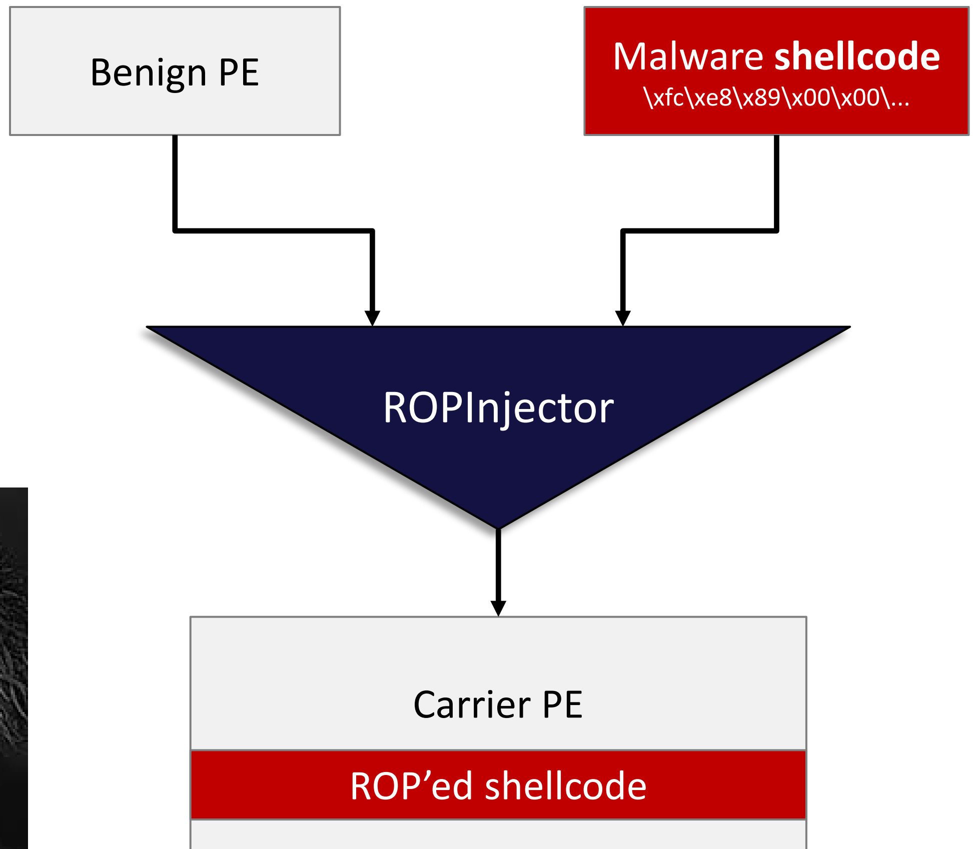


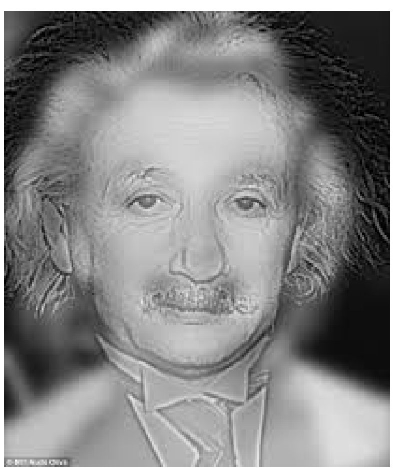
1 Portable Executable

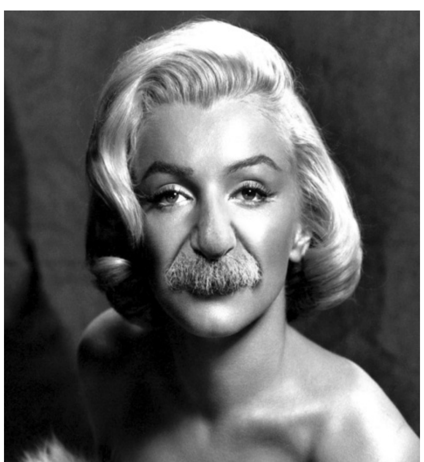
1 well-known shellcode

Many different variations

Our Tool: ROPInjector







Why use ROP for AV evasion?

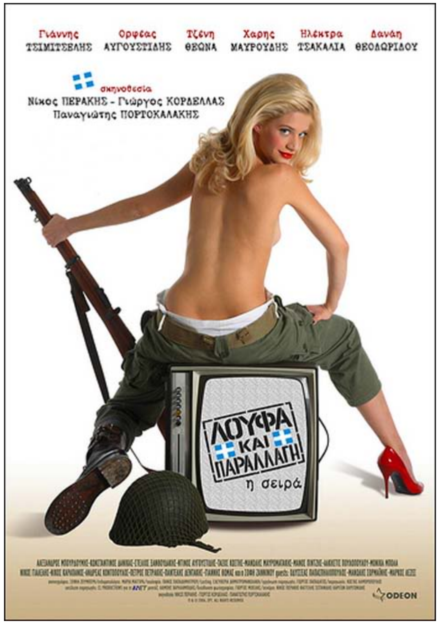
a) We use **borrowed code** (i.e., ROP gadgets)

➔ **Not raise any suspicious !**

- **A possible footprint:** the instructions that insert the **addresses of the ROP gadgets into the stack**.

b) May transform any given **shellcode** to a **ROP-based equivalent** ➔ **Generic**

c) May use **different ROP gadgets** or **the same found in different address** ➔ **Polymorphism**



A quick historical overview

plain malware code

\x59\xE8\xFF\x6B\x5F\xFF\x6A\x0F\x59\xE8\xFF

string signatures

\x6B\x5F\xFF\x6A\x0F

A quick historical overview

plain malware code → string signatures

simple obfuscation
(NOPs/dead-code in-between) → regex signatures

\x59\xE8\xFF\x6B\x5F\x90\xFF\x90\x6A\x0F\x59\xE8

variability

\x6B\x5F{\x90}*\xFF{\x90}*\x6A\x0F

black hat USA 2015

A quick historical overview

plain malware code → **string signatures**

simple obfuscation (NOPs/dead-code in-between) → **regex signatures**

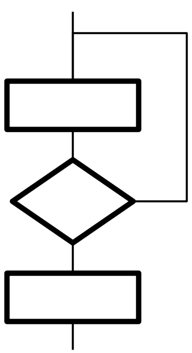
oligomorphism → **static analysis (disassembly, CFGs)**

PC ... PC

\x6A\x0F\x59\xE8\0xFF \x6B\x5F**************

decoder *encoded payload*

```
if RWX and performs
  then alarm
```



A quick historical overview

plain malware code	→	string signatures
simple obfuscation (NOPs/dead-code in-between)	→	regex signatures
oligomorphism	→	static analysis (disassembly, CFGs)
self-modifying code metamorphism	→	dynamic analysis (emulation, sandboxing, behavior-based signatures)

```
push eax  →  mov [esp-4],eax
             sub esp,4
```

Challenges for our Tool

1. The new **resulting PE** should **evade AV detection**

2. PE should **not be** **corrupted/damaged**

3. The **tool** should be **generic** and **automated**

4. Should **not require** a **writeable code section** to **mutate** (i.e., execute ROP chain)

Steps of ROPInjector

1. Analyze the shellcode
2. Find ROP gadgets in the PE
3. Transform the shellcode to an equivalent ROP chain
4. Inject into the PE missing ROP gadgets *(if required)*
5. Assemble a ROP chain building code in the PE
6. Patch the chain building code into the PE

STEP 1: Shellcode Analysis (1/3)

- Aims to obtain the **necessary information** to safely replace **shellcode instructions** with **gadgets**
- For each **instruction**, **ROPInjector** likes to know:
  - what **registers** it **reads**, **writes** or **sets**
  - what **registers** are **free** to modify
  - its **bitness** (a `mov al,X` or a `mov eax,X ?`)
  - whether it is a **branch** (`jmp`, `conditional`, `ret`, `call`)
    - and if so, where it **lands**
  - whether it is a **privileged** instruction (e.g., `sysenter, iret`)
  - whether it contains a **VA reference**
  - whether it uses **indirect addressing mode** (e.g., `mov [edi+4], esi`)

STEP 1: Shellcode analysis (2/3)

- **Scaled Index Byte (SIB)** enables **complex indirect addressing modes**

```
mov eax, [ebx+ecx*2]
```

- We want to avoid **SIBs** in the **shellcode** since
  - **long: >3 bytes**
    - unlikely to be found in gadgets
  - **rarely reusable**
  - **reserve at least 2 registers**

STEP 1: Shellcode analysis (3/3)

- **ROPInjector** transforms **SIB** into **simpler instructions**:

unrolling of SIBs

```
mov eax, [ebx+ecx*2]   →   mov eax, ecx
                           sal eax, 1
                           add eax, ebx
                           mov eax, [eax]
```

- ecx *is freed at this point*
- *shorter instructions*
- *reusable gadgets (either found or injected)*

- With **unrolling of SIBs,** we achieve:
  - **increased chances** of finding **suitable gadgets**
  - **less gadgets** being **injected**

STEP 2: Find ROP Gadgets in PE (1/2)

1. First, find **returns** of type:
   - `ret(n)` or
   - `pop regX`
     `jmp regX` or
   - `jmp regX`

2. Then, search **backwards** for more **candidate gadgets**

STEP 2: Find ROP Gadgets in PE (2/2)

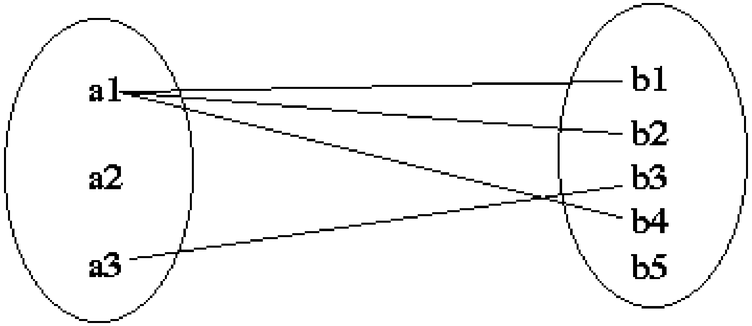
- **ROPInjector** automatically resolves **redundant instructions** in **ROP gadgets**
  - Avoid **errors** during the execution of **ROP code**
- Maximize **reusability** of **ROP gadgets**
- Avoid **injecting unsafe** **ROP gadgets**
  - modify **non-free registers**
  - are **branches**
  - write to the **stack** or modify **esp**
  - are **privileged**
  - use **indirect addressing mode**

STEP 3: Transform shellcode to ROP chain

- Initially, it translates **shellcode instructions** to an **Intermediate Representation** (IR).
- Next, it translates the **ROP gadgets** found in PE to an **IR.**
- Finally, it provides a **mapping between** the two **IRs**
  - **1 to 1**

  or

  - **1 to many**



black hat USA 2015

STEP 3: Intermediate Representation

IR Type (20 in total)	Semantics	Eligible instructions

ADD_IMM	regA += imm	add r8/16/32, imm8/16/32
		add (e)ax/al, imm8/16/32
		xor r8/16/32, 0
		cmp r8/16/32, 0
		inc r8/16/32
		test r_a32, r_b32 (with r_a == r_b)
		test r8/16/32, 0xFF/FFFF/FFFFFFFF
		test (e)ax/al, 0xFF/FFFF/FFFFFFFF
		or r_a32, r_b32 (with r_a == r_b)

MOV_REG_IMM . . .	mov regA, imm	mov r8/16/32, imm8/16/32 imul r16/32, r16/32, 0 xor r_a8/16/32, r_a8/16/32 and r8/16/32, 0 and (e)ax/al, 0 or r8/16/32, 0xFF/FFFF/FFFFFFFF or (e)ax/al, 0xFF/FFFF/FFFFFFFF

STEP 3: Mapping examples

- 1-1 **mapping** example
  - **Shellcode:**

    `mov eax, 0` → MOV_REG_IMM(eax, 0)

  - **Gadget in PE:**

    ```
    and eax, 0
    ret
    ```
    → MOV_REG_IMM(eax, 0)

    1 to 1 IR mapping

- 1-many **mapping** example
  - **Shellcode:**

    `add eax, 2` → ADD_IMM(eax, 2)

  - **Gadget in PE:**

    ```
    inc eax
    ret
    ```
    → ADD_IMM(eax, 1)

    1 to 2 IR mapping

STEP 4: Gadget Injection

- If the **PE** does **not include** the **required ROP gadgets**
- By simply injecting **ROP gadgets** would raise **alarms**

Statistics (presence of successive ret instructions)

- Therefore, we insert **ROP gadgets** in a **benign looking way** (**scattered**) avoiding alarms:
  - `0xCC` caves in **.text section** of PEs *(padding space left by the linker)*
  - Often preceded by a **ret** *(due to function epilogue)*

```
00000640  FC 1E 00 00 E9 19 31 00  00 E9 44 09 00 00 CC CC  ......1...D....
00000650  CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ...............
00000660  CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ...............
00000670  CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ...............
```

STEP 4: Gadget Injection

- Assuming the **missing gadget** is **`mov ecx, eax`** and we find the following `0xCC` cave:

```
    <other instructions>

epilogue:
    mov esp, ebp
    pop ebp
return:
    ret(n)
    CCCCCCCCCCCCCCCCCCCCCC
```

STEP 4: Gadget Injection

- Assuming the **missing gadget** is `mov ecx, eax` and we find the following `0xCC` cave:

```
    <other instructions>
    jmp epilogue
    mov ecx, eax
    jmp return
epilogue:
    mov esp, ebp
    pop ebp
return:
    ret(n)
    CCCCCCCC
```

STEP 4: Gadget Injection

- Assuming the **missing gadget** is **`mov ecx, eax`** and we find the following `0xCC` cave:

Normal flow

```
<other instructions>
jmp epilogue
mov ecx, eax
jmp return
epilogue:
mov esp, ebp
pop ebp
return:
ret(n)
CCCCCCCC
```

STEP 4: Gadget Injection

- Assuming the **missing gadget** is **mov ecx, eax** and we find the following 0xCC cave:

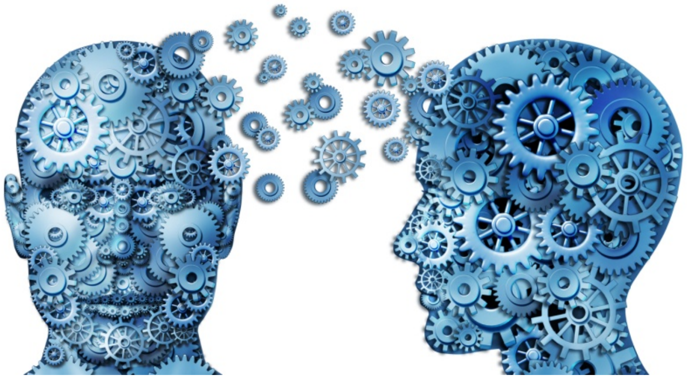
Normal flow

```
  <other instructions>
  jmp epilogue
  mov ecx, eax
  jmp return
epilogue:
  mov esp, ebp
  pop ebp
return:
  ret(n)
  CCCCCCCC
```

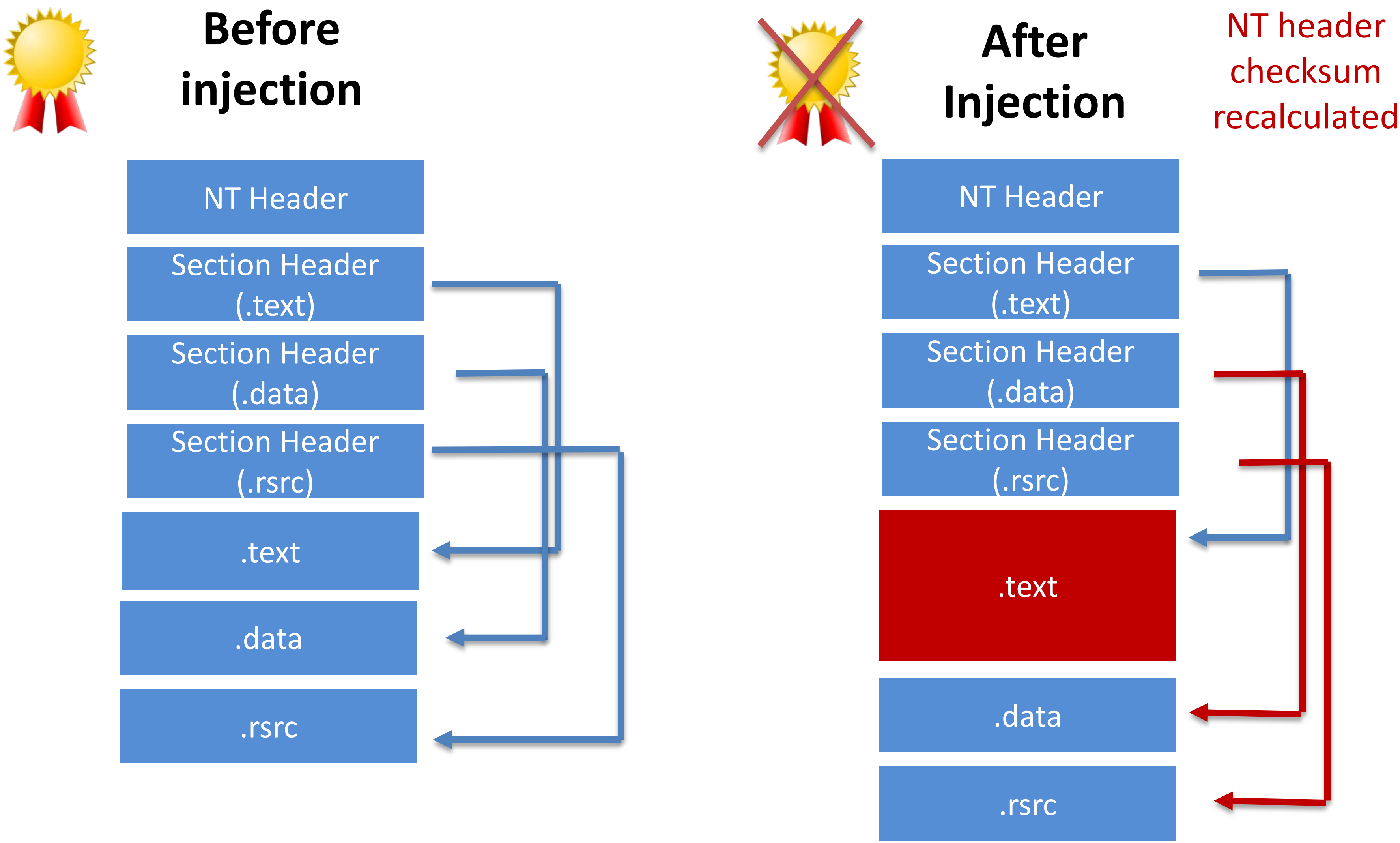
ROP flow

STEP 5 and 6: Assemble and patch the ROP chain into the PE

- **Step 5:** Insert the **code** that **loads** the **ROP chain** into the **stack** *(mainly PUSH instructions)*
- **Step 6 patch the new PE:** Extends the **.text section** (instead of adding a new one), and, then, **repair** all **RVAs** and **relocations** in the **PE**.
- **ROPInjector** includes **two** different methods to **pass control** to the **ROPed shellcode**
  - **Run first**
  - **Run last**

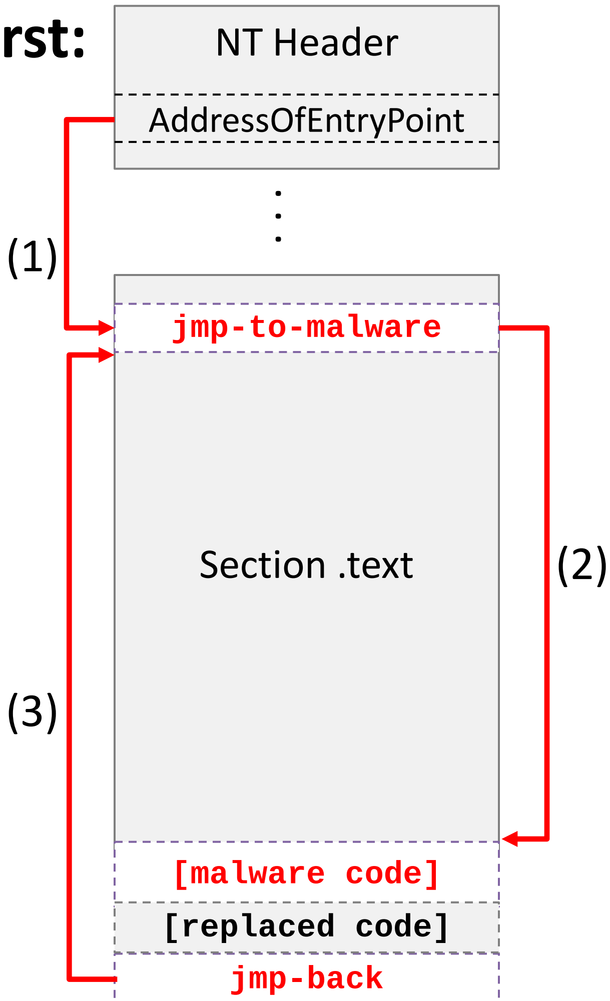


STEP 6: PE Patching (1/2)



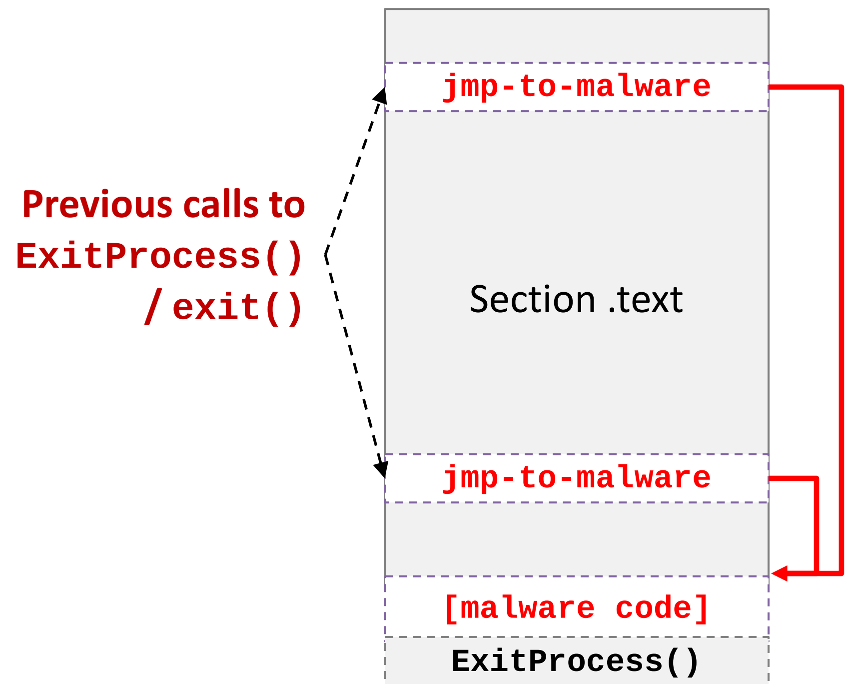
STEP 6: PE Patching (2/2)

Run first:



Run last:

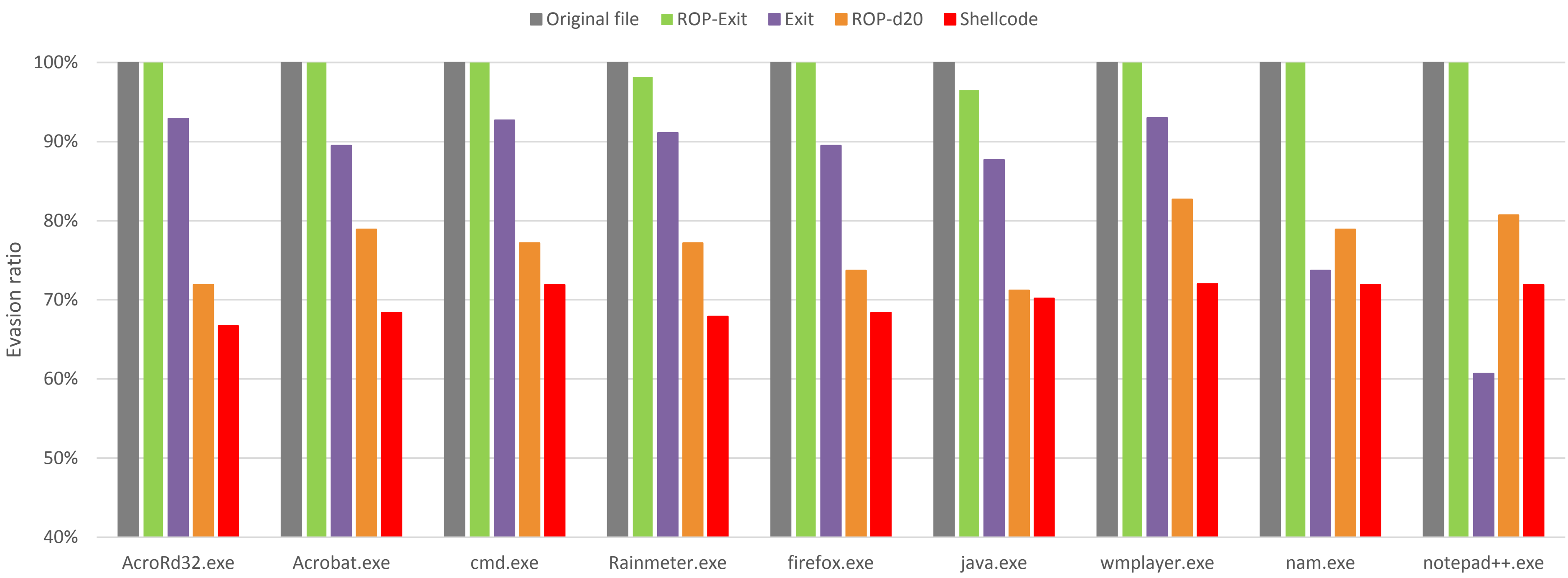
(very good anti-emulation results)



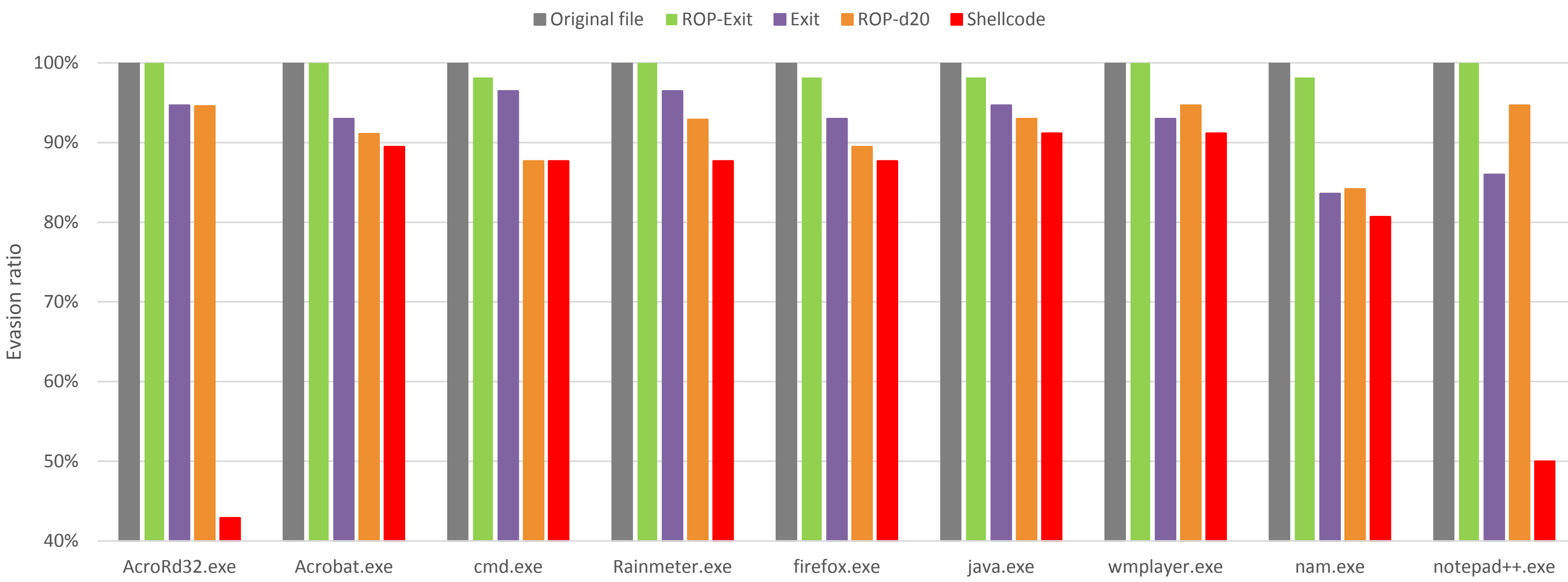
Evaluation

- ROPInjector is implemented in **native Win32 C**
- Nine (9) **32bit Portable Executables**
  - firefox.exe, java.exe, AcroRd32.exe, cmd.exe, notepad++.exe and more
- Various combinations – scenarios
  - **Original-file (no patching at all)**
  - **ROPShellocode-Exit (ROP’ed shellcode and run last)**
  - **Shellcode-Exit (intact shellcode passed control during exit)**
  - **ROPShellcode-First-d20 (ROP’ed shellcode and delayed execution, 20 secs)**
  - **Shellcode (intact shellcode)**
- 2 of the most popular **Metasploit payloads**
  - **reverse TCP shell**
  - **meterpreter reverse TCP**
- VirusTotal
  - at the time it employed **57 AVs**

Evasion rate: reverse TCP shell

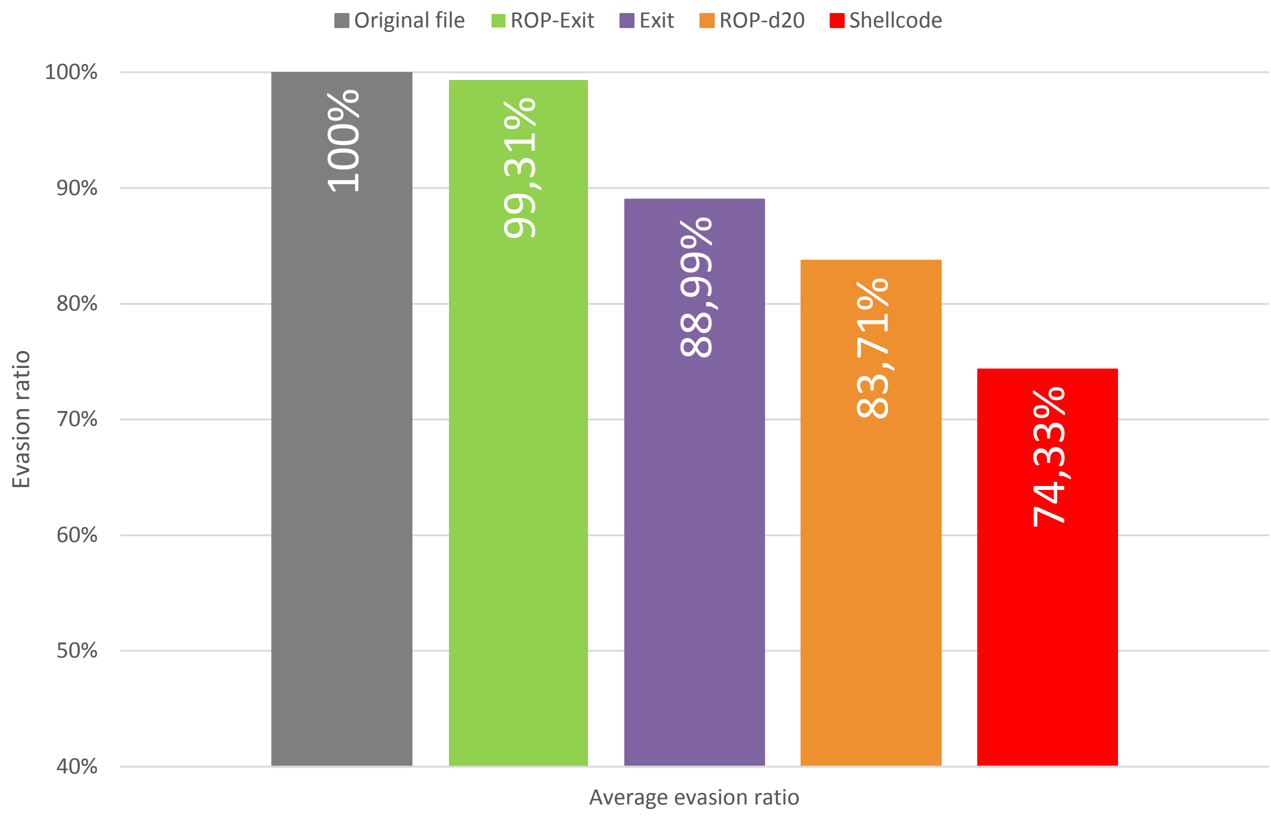


Evasion rate: meterpreter reverse TCP



Overall evasion results

- **100% most of the times**
- **99.31% on average**



Outcomes

- **Signature-based detection** can be bypassed by techniques like **ROP'ed shellcodes**
- **Behavioral analysis** can also be bypassed by techniques like **running right before process exit**
- **Checksums** and **certificates** provide **poor protection**

Proposals

- Engagement of certificates and checksums
- Enhancement of behavioral analysis
- Execution of behavior analysis until the program <u>really</u> ends

Thank you!

Questions?

Prof. Christos Xenakis

Systems Security Laboratory, Department of Digital Systems,

School of Information and Communication Technologies,

University of Piraeus, Greece

http://ssl.ds.unipi.gr/

http://cgi.di.uoa.gr/~xenakis/

email: xenakis@unipi.gr