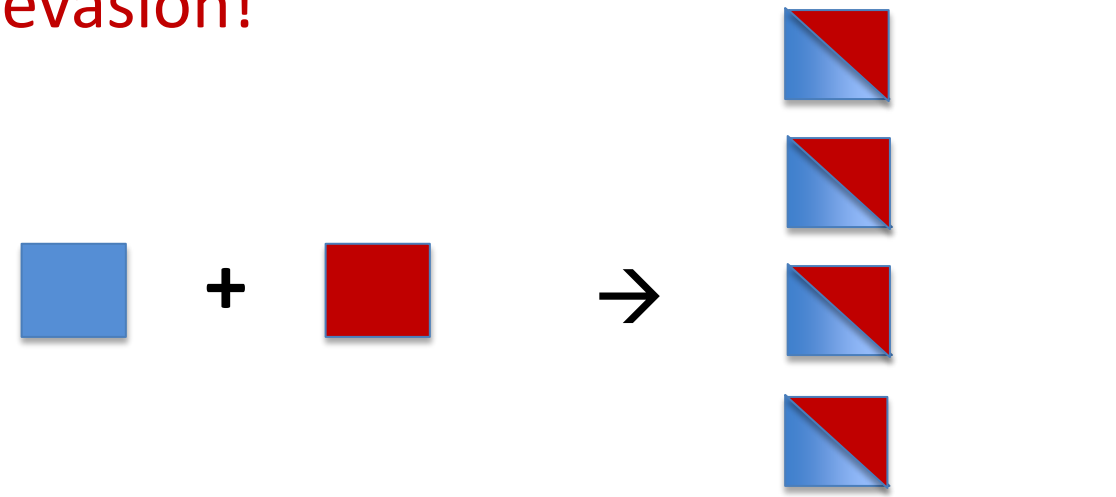
ROPInjector: Using Return-Oriented Programming for Polymorphism and AV Evasion

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

UBM Tech

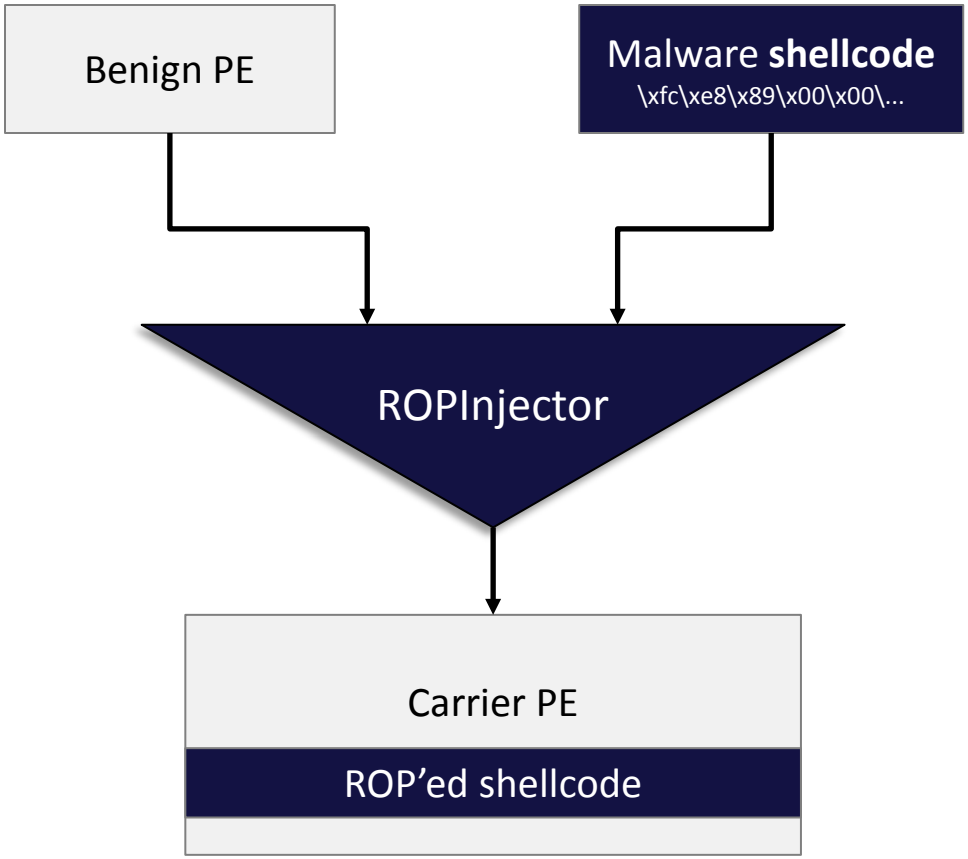
Overview

- Return Oriented Programming (ROP) has been used in the past explicitly for DEP evasion in software exploitation scenarios.
- We propose a completely different use of ROP!
- We propose ROP as a polymorphic alternative to achieve AV evasion!



ROPInjector

- Local infection of benign PE executables with well-known alarming malicious code (i.e., shellcode)



Why ROP for AV evasion?

- Does not raise suspicious → Borrowed code (i.e., rop gadgets) is of course benign. The only possible detection footprint is the instructions (i.e., typically push) that insert the addresses of the ROP gadgets into the stack.

- Generic → ROP can be used to transform any given malware shellcode to a ROP-based equivalent.

- Polymorphism → Use different ROP gadgets or use same ROP gadgets found in different address

Challenges

1. The new resulting PE should evade antivirus detection
2. The benign PE should not be corrupted/damaged
3. The tool should be generic and completely automated
4. Should not require a writeable code section to mutate (i.e., execute ROP chain)

Challenges Accepted!

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

simple obfuscation (NOPs/dead-code in-between)

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

variability

string signatures

regex signatures

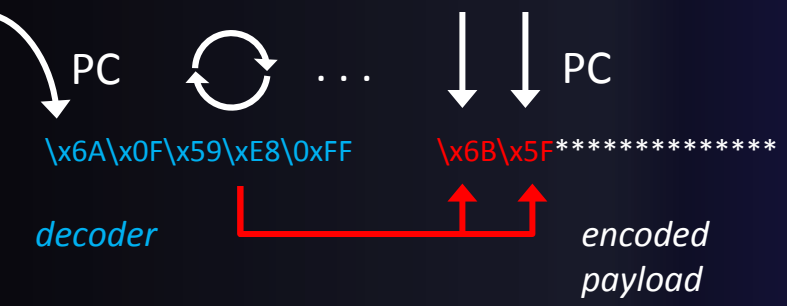
\x6B\x5F{\x90}*\xFF{\x90}*\x6A\x0F

A quick historical overview

plain malware code

simple obfuscation (NOPs/dead-code in-between)

oligomorphism

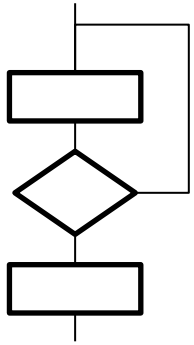


string signatures

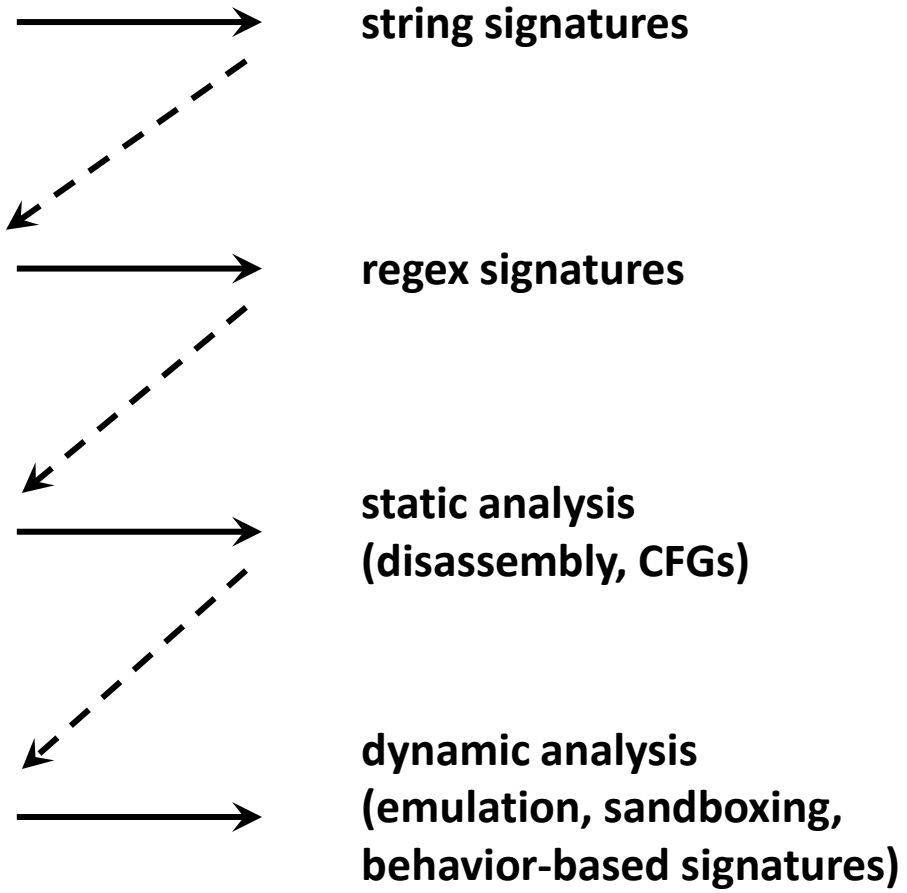
regex signatures

static analysis (disassembly, CFGs)

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



A quick historical overview




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

ROPInjector Steps

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

STEP 1: Shellcode Analysis (1/3)

- The analysis of the shellcode aims to obtain various information so that it can safely replace shellcode instructions with gadgets

- In particular, 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`)

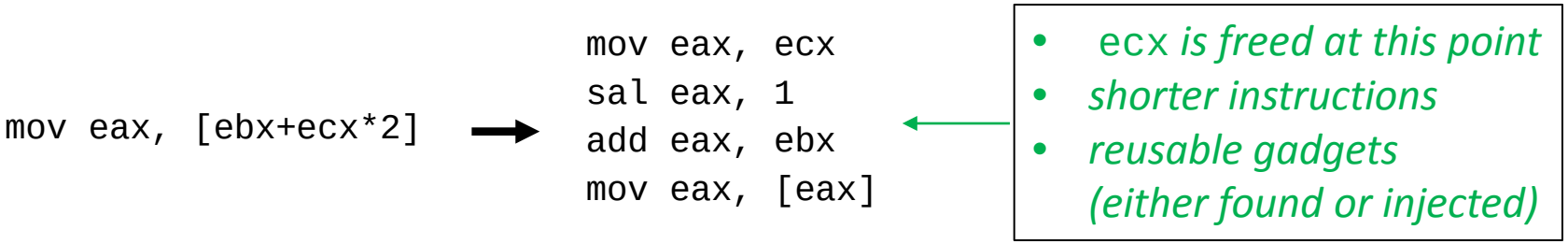
STEP 1: Shellcode analysis (2/3)

- Scaled Index Byte (SIB) enables complex indirect addressing modes
  - e.g. `mov [eax+8*edi+10], ecx`

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

STEP 1: Shellcode analysis (3/3)

- ROPInjector performs a technique that we call it **Unrolling of SIBs** to transform them into simpler instructions



- With this technique 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`
     - used only if a "loader" gadget of the following form is also found:

       ```
       pop regX
       [any of the first 2 endings above]
       ```

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, in order to avoid errors during 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

- First, we translate shellcode instructions to an Intermediate Representation (IR).
- Next we translate ROP gadgets found in PE to an IR.
- Finally, a mapping is performed between the two IRs
  - 1 to 1
  - 1 to many

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:

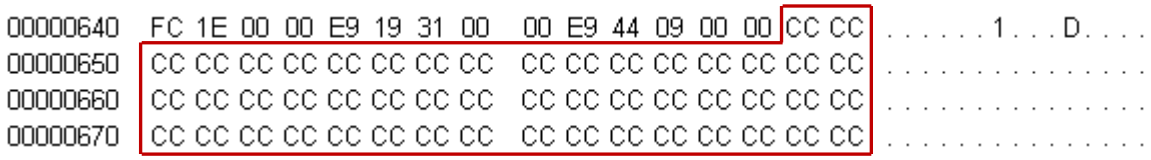
```
inc eax
ret
```

→ ADD_IMM(eax, 1)

1 to 2 IR mapping

STEP 4: Gadget Injection

- In some cases PE does not include the required ROP gadgets
- Simply injecting ROP gadgets would raise alarms using statistics
  - Presence of successive ret instructions
- To this end, ROPInjector inserts ROP gadgets scattered, and in a benign looking way to avoid alarms
  - 0xCC caves in .text section of PEs (padding space left by the linker)
  - Often preceded by a RET (due to function epilogue)



STEP 4: Gadget Injection

- Assuming 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
```

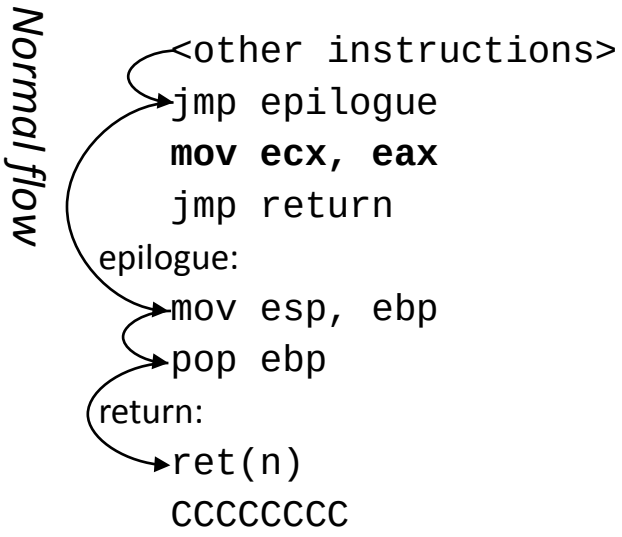
STEP 4: Gadget Injection

- Assuming 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
```

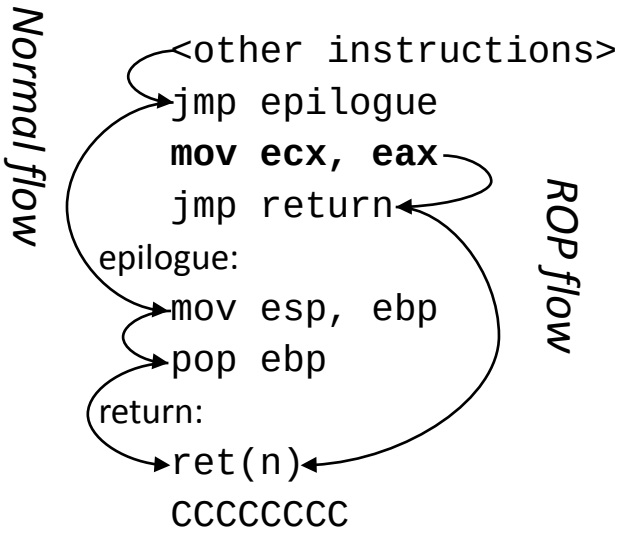
STEP 4: Gadget Injection

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



STEP 4: Gadget Injection

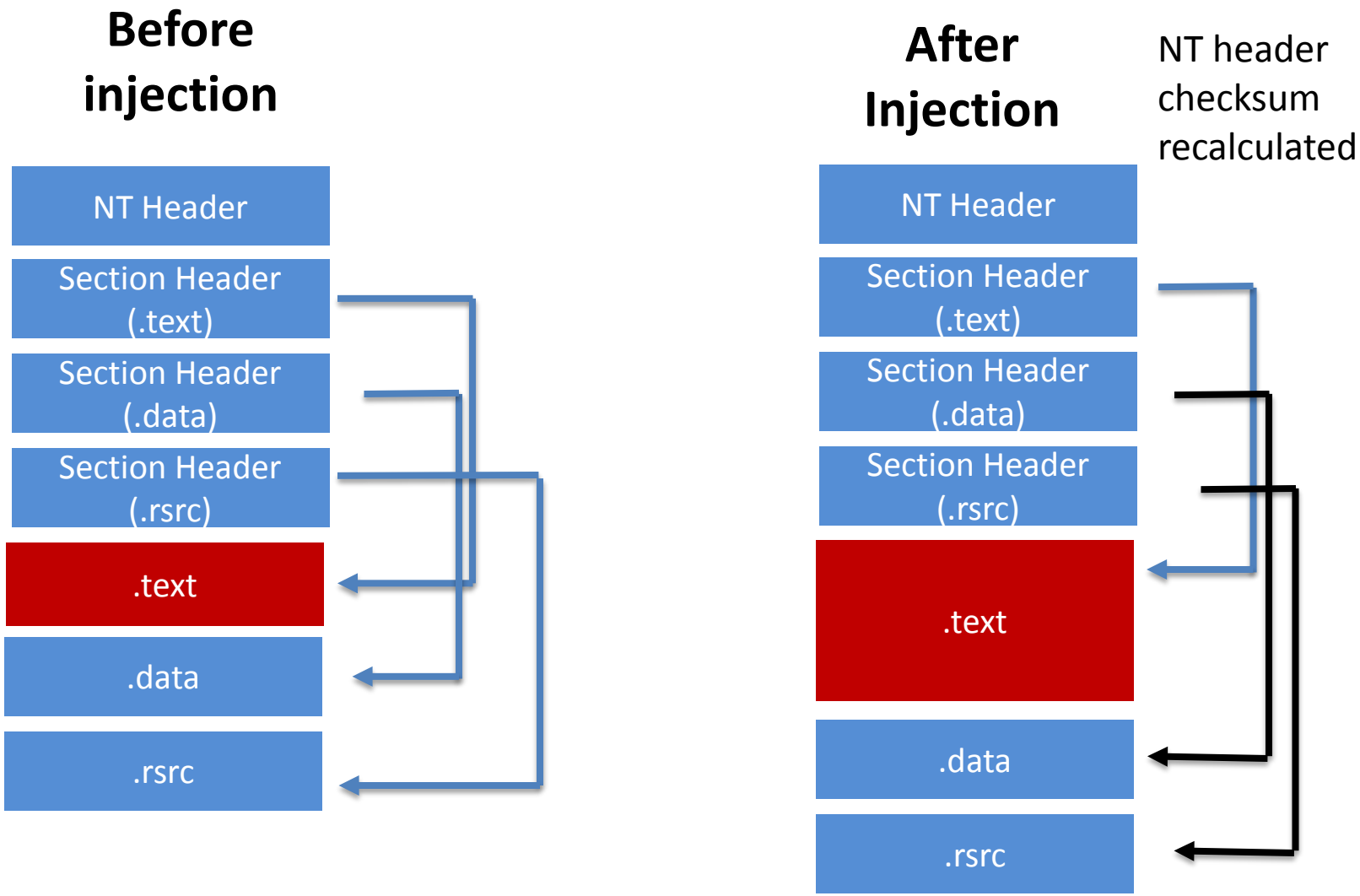
- Assuming missing gadget is **`mov ecx, eax`** and we find the following `0xCC` cave:



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

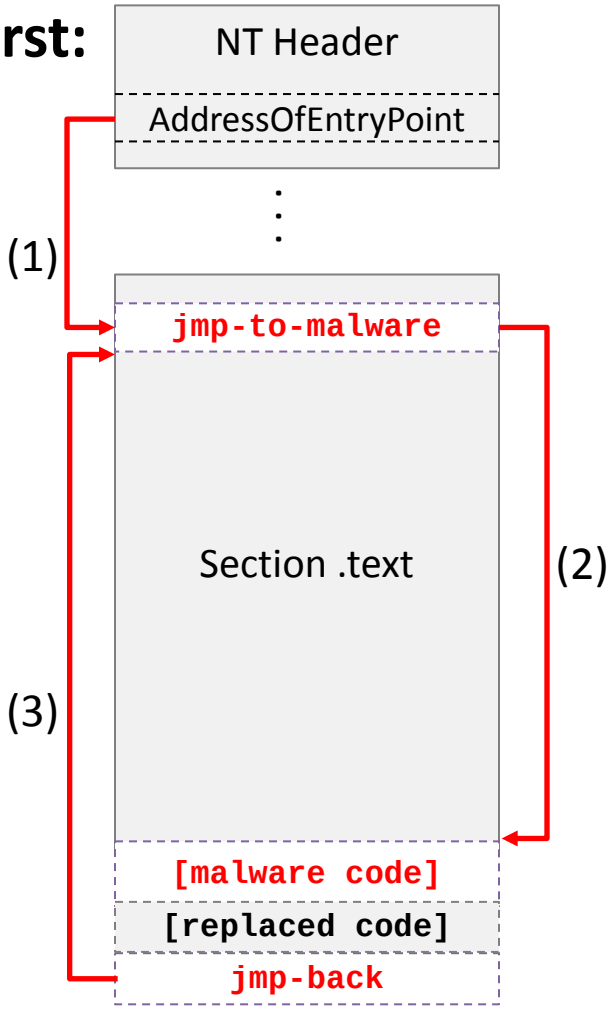
- In step 5 we insert the code that loads the ROP chain onto the stack (mainly PUSH instructions)
- In step 6 we patch the new PE → ROPInjector extends the .text section (instead of adding a new one that would raise alarm) and then goes on to repair all RVAs and relocations in the PE.
- ROPInjector includes two different methods to pass control to the ROPed shellcode
  - Run first
  - Run last

STEP 6: PE Patching (1/2)



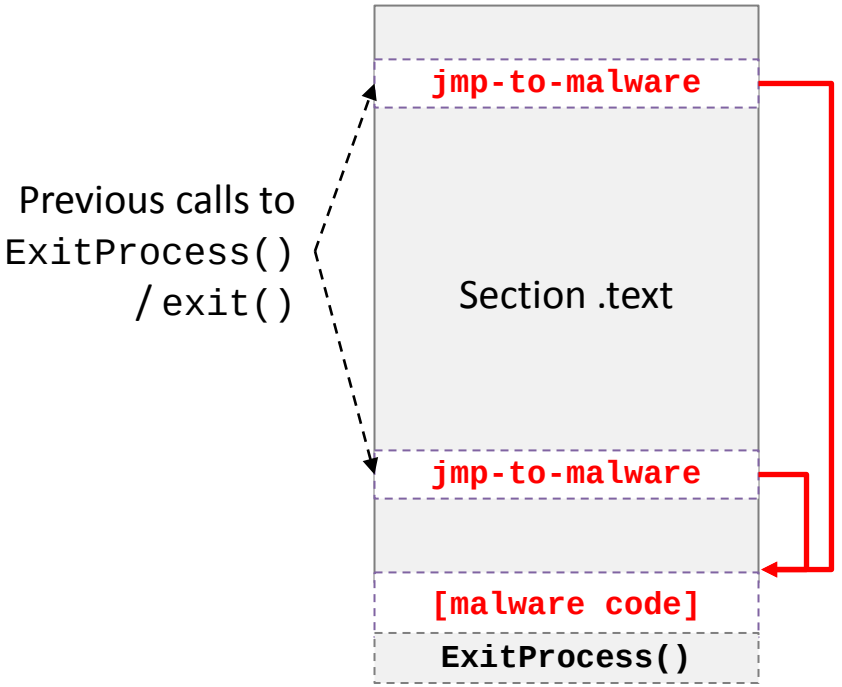
STEP 6: PE Patching (2/2)

Run first:



Run last:

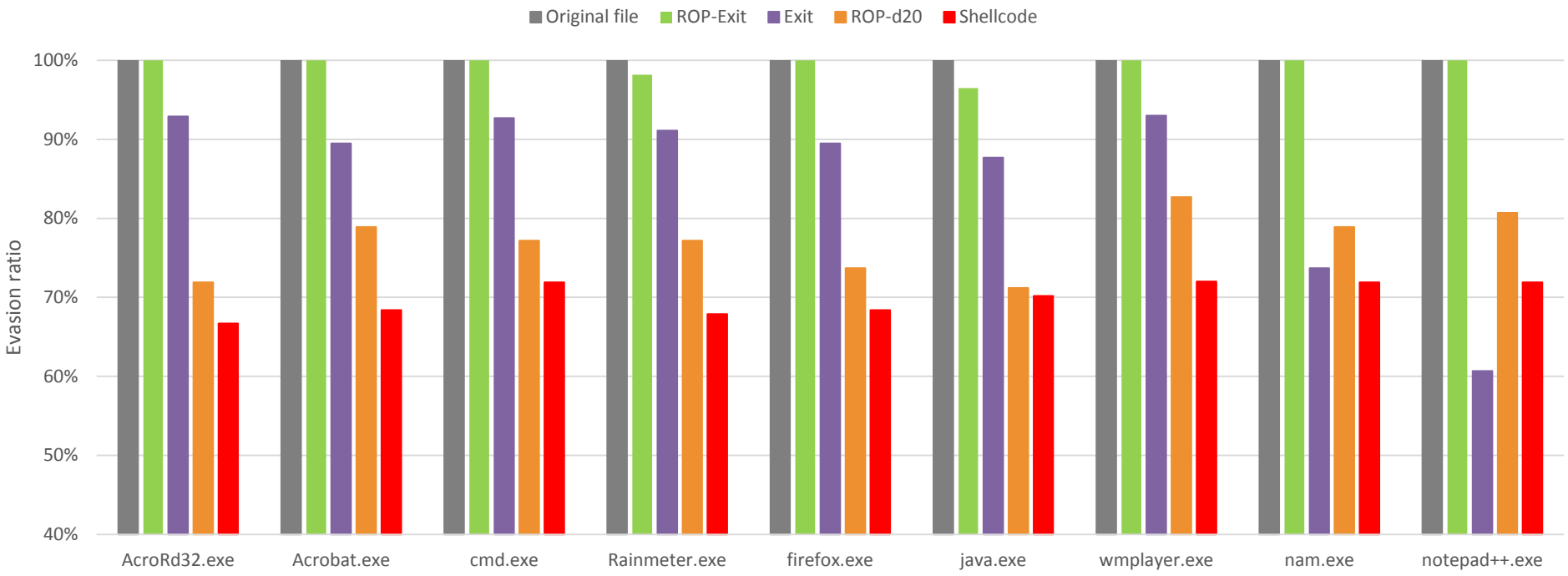
(very good anti-emulation results)



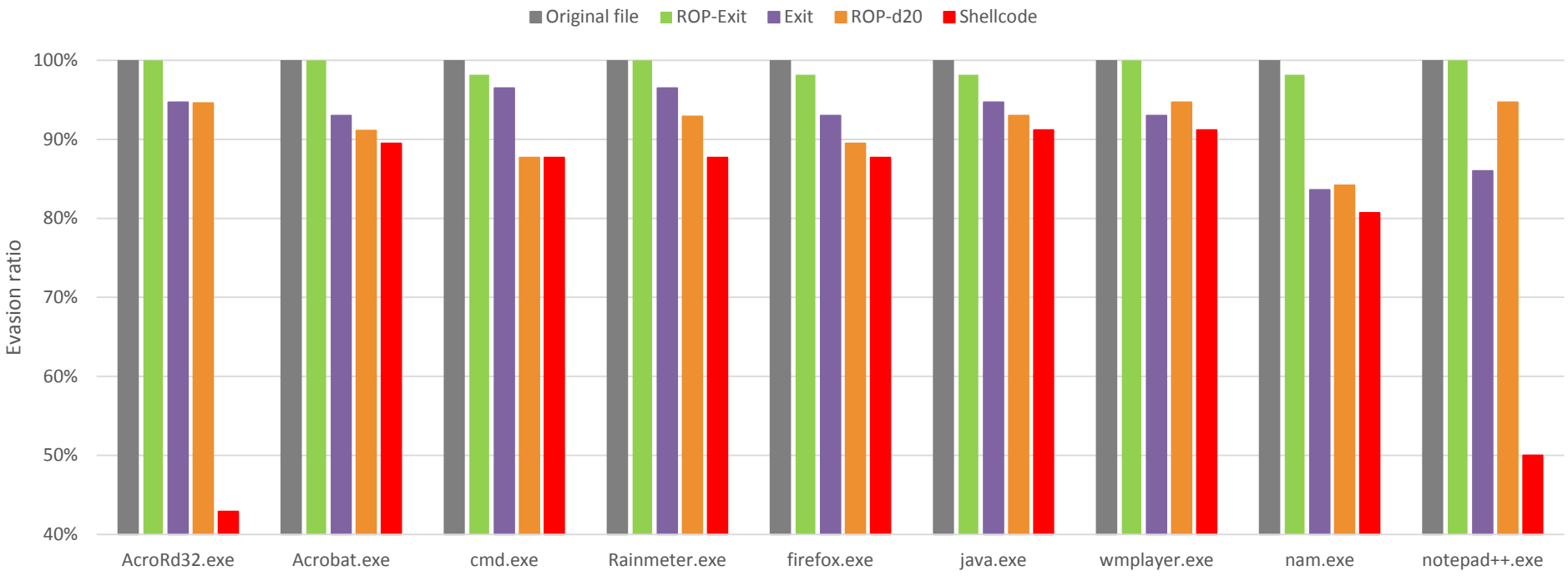
Evaluation

- Implemented in native Win32 C
- 9 32bit executables
  - firefox.exe, java.exe, AcroRd32.exe, cmd.exe, notepad++.exe and more
- Various combinations
  - No patching at all
  - ROP'ed shellcode and run last
  - Intact shellcode passed control during exit (run last)
  - ROP'ed shellcode and delayed execution (20 secs via Win32 Sleep())
  - Intact shellcode
- 2 of the most popular payloads of MSF
  - reverse TCP shell
  - meterpreter reverse TCP
- VirusTotal
  - at the time it employed 57 AVs

Evasion rate: reverse TCP shell

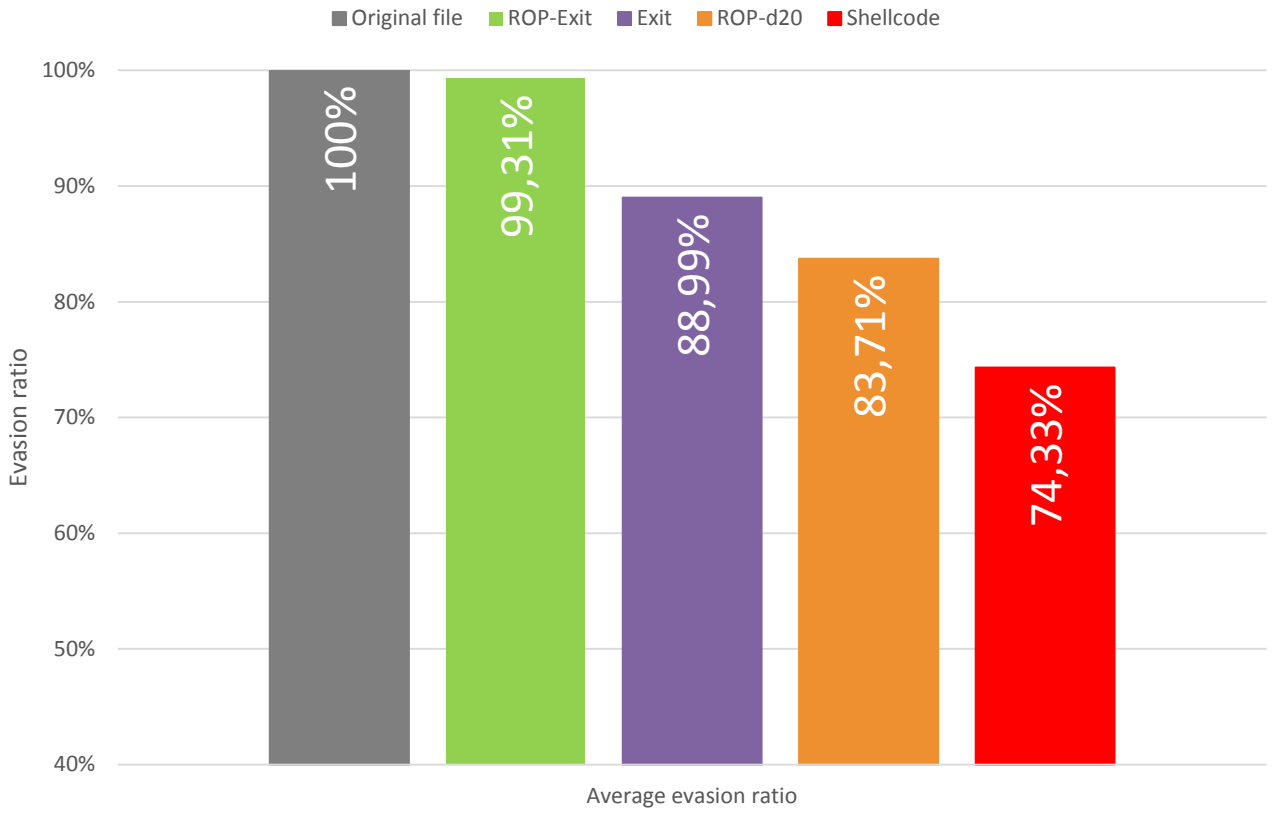


Evasion rate: meterpreter reverse TCP



Overall evasion results

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



Black Hat Sound Bytes

- Current signature-based detection methods are no longer effective
  - we shown that by using ROP we can reduce the footprint to benign stack modifying instructions
- Behavioral analysis is tough to perform exhaustively
  - we shown how to easily bypass it by running right before process exit
- “Default distrust all” policy
  - Checksums and certificates is the poor user’s last line of defense at the moment