Yataglass: Network-level Code Emulation for Analyzing Memory-scanning Attacks

Makoto Shimamura (1),
Kenji Kono (1,2)

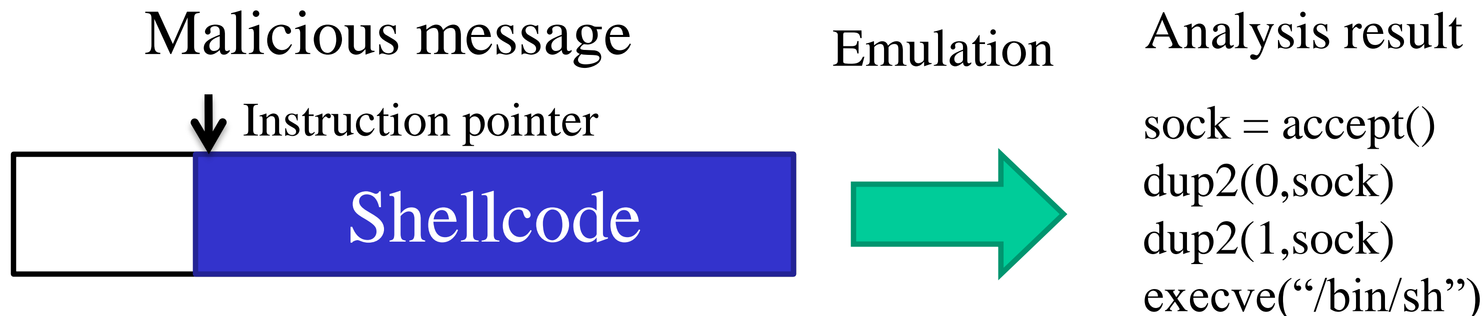
(1) Dept. of Information and Computer Science, Keio Univ., Japan
(2) CREST, Japan Science and Technology Agency

Remote code injection attack

- Allows attackers to execute their arbitrary shellcode
  - Various vulnerabilities can be exploited
    - Stack overflow, Heap overwrite, Format string attack etc…
- Security researchers analyze shellcode to develop countermeasures
  - Static disassembly is widely used
- Attackers can thwart static disassembly
  - Encryption
    - encrypts shellcode body
  - Obfuscation
    - inserts junk bytes between instructions

Network-level code emulator

- Emulate the execution of shellcode
  - e.g.) Spector [Borders, *et al.* '07]
    - extracts system functions issued by shellcode
- Advantage
  - Never thwarted by encryption and obfuscation
    - Encrypted shellcode is decrypted during execution
    - Obfuscation cannot hide the presence of system call invocations

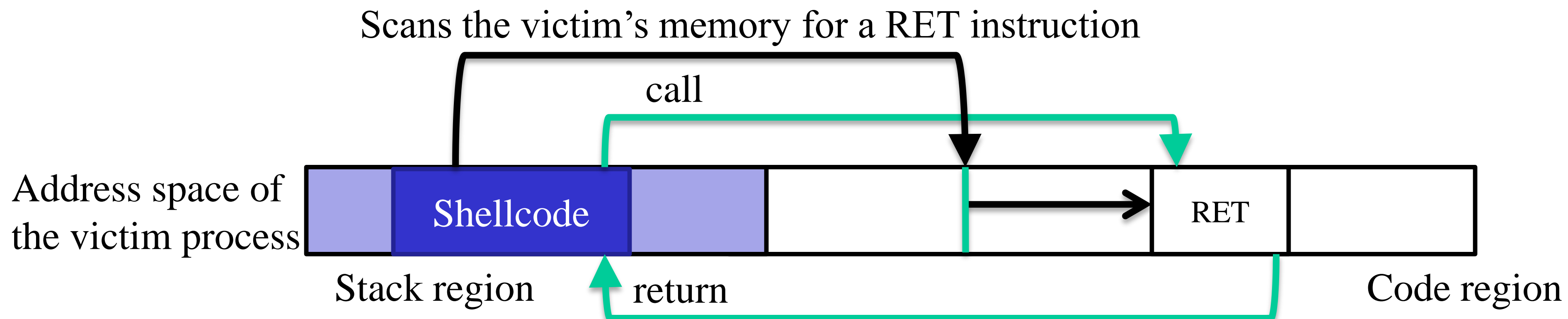


Typical application of network-level code emulator

- Analyze shellcode collected by honeypots
  - Honeypot is a decoy host that collects malicious network traffic
    - Allows us to collect a lot of shellcode for various servers
  - Many anti-virus vendors, security research institutes have their honeypots
- Network-level code emulators extract executed instructions and system calls of collected shellcode
  - The result is used for…
    - Behavior-based virus detection of anti-virus software
    - Restoring compromised servers from damage

Memory-scanning attack

- Memory-scanning attack can evade network-level code emulators
  - ►Uses instructions in the victim process, that are outside shellcode, as a part of shellcode
- Current network-level code emulators cannot analyze shellcode of this style
  - ►No emulator uses the victim's memory for emulation

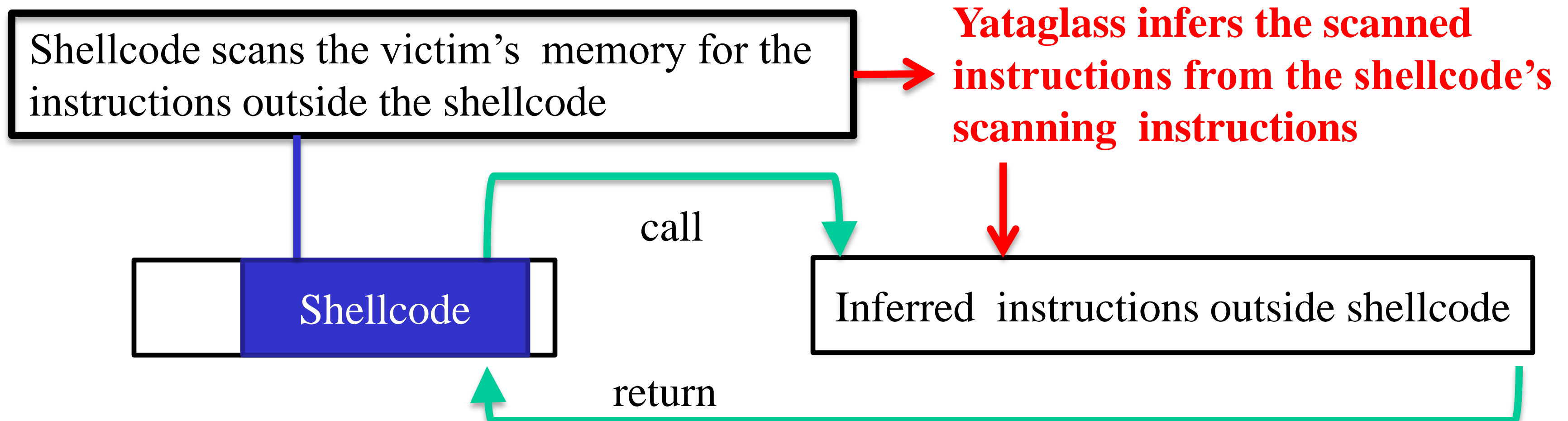


Why not use the victim's memory image?

- Using the victim's memory image is cumbersome
  - ►In particular, when that honeypots collect shellcode…
    - The analyst must prepare memory images of possible targeted software and their various versions
    - No real victim process exists if the honeypot is low-interaction honeypot
- It is better to analyze shellcode without victim process's memory
  - ►Enables us to analyze the shellcode collected by honeypots with less burden
  - ►No need to prepare many memory image

Proposal : Yataglass

- A network-level code emulator that allows us to analyze memory-scanning attack
  - ▶Infers instructions outside shellcode that a memory-scanning shellcode scans for
- Victim's memory image is not required
  - ▶Enables us to analyze memory-scanning attack effectively



Scanning loop

- A ***scanning loop*** scans the victim's memory for instructions
  - Example : scans for a RET instruction (0xC3)

```
1:              mov  edi, ADDR    (An addr. of the victim proc.)
2:  LOOP:       inc  edi
3:              cmpb [edi], 0xC3
4:              jne  LOOP
5:              call  edi          # Uses the found RET
6:  CONTINUE:  # Shellcode continues
```

Scanning loop for 'RET'

- Yataglass infers what instructions are scanned for
  - Infers the instructions from the exit-condition of the scanning loop
  - In this example, EDI register points to a RET instruction when the control exits from the scanning loop

Symbolic execution

- To infer the scanned-for instructions, Yataglass uses symbolic execution
- Symbolic execution executes a program without concrete values
  - ►Values are regarded as symbols
  - ►Operations are done symbolically
  - ►A result of an operation is expressed as a new symbol that contains operator and operands

Instruction sequence	Symbolic execution	
mov eax, INPUT1	eax = X	← A symbol for unknown INPUT1
mov ebx, INPUT2	ebx = Y	← A symbol for unknown INPUT2
add eax, ebx	eax = (X + Y)	

Inferring scanned-for instructions by symbolic execution

- Yataglass forks if an unknown symbol is used as a predicate of conditional branch
  - ► Executes both branch with appropriate constraints
  - ► The instance of Yataglass which exits from the loop has appropriate conditions to exit from the scanning loop
  - ► Yataglass terminates execution if the same loop is executed to prevent path explosion

	Instruction sequence	**Symbolic execution**
1:	mov edi, X	edi = X (An addr. of the victim proc.)
2:	LOOP: inc edi	edi = (X + 1)
3:	cmpb [edi], 0xC3	Compared *(X+1) with 0xC3
4:	jne LOOP	Set constraint [edi] == 0xC3
5:	call edi	Jump to [edi] == RET

More complicated scanning

- Using multiple constraints to find an instruction

		Instruction sequence	Symbolic execution
1:		mov edi, X	edi = X (An addr.of the victim proc.)
2:	LOOP:	inc edi	edi = X + 1
3:		cmpb [edi], 0xC2	Compared *(X+1) with 0xC2
4:		jle LOOP	Set constraint [edi]>0xC2
5:		cmpb [edi], 0xC4	Compared *(X+1) with 0xC4
6:		jge LOOP	Set constraint [edi]<0xC4
7:		call edi	([edi]>0xC2) && ([edi]<0xC4) -> [edi] == 0xC3 ('RET')

Experiment: Analysis of memory-scanning attacks

- Obtained seven realworld shellcode from SecurityFocus and Milw0rm
- Inserted memory-scanning code to the shellcode
- Compared execution result with Spector [Borders, et al., '07]
  - ► Spector is one of the state-of-the art network-level code emulator

Source	**Target**	**Obtained from**	**Yataglass**	**Spector**
tsig.c	bind	SecurityFocus	✓	✘
7350wurm.c	wu-ftpd	MilwOrm	✓	✘
rsync-expl.c	rsync	SecurityFocus	✓	✘
7350owex.c	wu-imap	MilwOrm	✓	✘
OpenFuck.c	Apache	SecurityFocus	✓	✘
sambal.c	Samba	SecurityFocus	✓	✘
cyruspop3d.c	Cyrus-pop3d	MilwOrm	✓	✘

Analysis result of real shellcode

- Analyzed shellcode for B/O vuln. in samba 2.2.7 that incorporates memory-scanning code
- Yataglass extracted a list of system calls issued by the shellcode and that of executed instructions

Issued system calls

```
SOCK1=socket(2,1,6)
listen(SOCK1,{2,61360,0},16)
SOCK2=accept(SOCK1,0)
close(SOCK1)
dup2(SOCK2,0)
dup2(SOCK2,1)
execve("/bin//sh","/bin//sh")
```

Executed instructions (snippet)

```
…
push esi
push ebp
jmp edi
pop ebp
ret
popa
int 0x80
…
```

Analysis result of real shellcode (cont.d)

- We manually analyzed the shellcode by injecting it into the target server and tracing instructions with GDB
  - ►accepts a network connection from the attacker by socket(), listen() and accept()
  - ►redirects the stdin/out to the connection by dup2()
  - ►executes /bin/sh by execve()
- Confirmed the result generated by Yataglass

Limitations

- Yataglass cannot infer instructions if the shellcode scans for a value in a range
  - pop instructions ranges from 0x58 to 0x5F regarding registers
    - pop eax=0x58, pop ebx=0x59, … pop edi = 0x5F
  - Shellcode may use a scanning loop that accepts all pop instructions followed by ret instruction
    - e.g.) save all registers, push garbage value, call the scanned pop and ret, and then restore registers
  - Solution: fork() with assuming one of the possible values
- Yataglass cannot infer instructions when shellcode scans for a function signature
  - Shellcode may scan for the first several bytes of fopen() to invoke it
  - We think signature-based inference is useful

Related work

- Spector [Borders, et al. ’07]
  - ► Uses symbolic execution to extract behaviors of shellcode
  - ► Can be evaded by memory-scanning attacks
- Detection of decryption behavior in polymorphic shellcode using emulation [Polychronakis, et al. ’06]
  - ► Counts payload reads followed by GetPC code
  - ► Can be evaded by memory-scanning attacks
    - But we can easily apply Yataglass’s technique to this emulator
- Polymorphic worm detection based on static analysis [Kruegel, et al. ’05]
  - ► Extracts possible control flows inside payloads and finds a match between extracted control flows in multiple streams
  - ► Yataglass extracts detailed behavior of shellcode used by worms

Summary

- Memory-scanning attack
  - ►Uses instructions of the victim process as a part of shellcode
  - ►Evades current network-level code emulators
- Proposed Yataglass to analyze memory-scanning attacks
  - ►Infers the scanned-for instructions with symbolic execution
  - ►Successfully analyzed memory-scanning shellcode without victim process's memory image
- Future work
  - ►Automatic defense against shellcode
  - ►Automatic recovery from the damage of shellcode