Audit: Intrusion Detection and Antivirus

CS 491/591

Fall 2015

The Three A's

- Authentication
- Authorization
- Audit
  - Maybe something went wrong
  - What *really* happened?

  - Maybe we want to prevent future problems
  - What's *really* going on in the system?

Audit: Historical Perspective

- In the 90's
  - Tech was booming; Rapid growth, rapid change
  - Software vendors didn't care about security
  - Choices were limited; Open source was new
  - System security was weak
  - Networks were growing faster, more pervasive
  - Attackers were getting smarter, more numerous

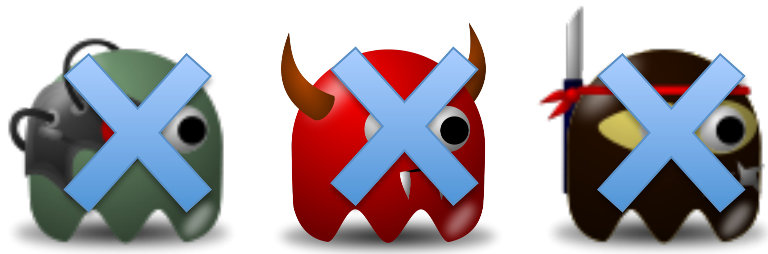
- What would you do in this situation?

Approaches for Audit

- Blacklisting
- Anomaly Detection
- Whitelisting

Approaches for Audit

- Blacklisting



- Anomaly Detection
- Whitelisting

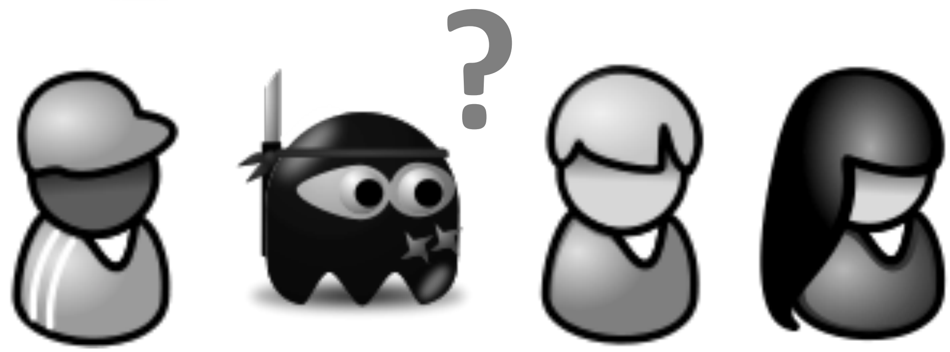
Approaches for Audit

- Blacklisting
- Anomaly Detection
- Whitelisting

Approaches for Audit

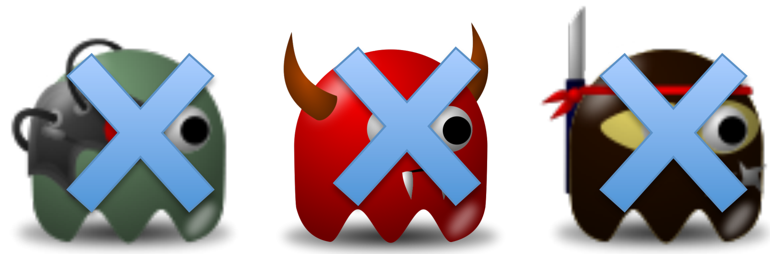
- Blacklisting

- Anomaly Detection



- Whitelisting

Approaches for Audit

- Blacklisting 

- Anomaly Detection

- Whitelisting

Blacklisting

- Idea: Disallow any known bad behaviors
- Example: Anti-Virus
  - Maintain a list of signatures of all known malicious software
  - Periodically check all files on the system against these signatures
  - Raise an alarm if found

Signature Example: Homework 2

- Say we want to detect attacks on guessing game programs
- How could we do it?

Signatures: Idea 1

- Idea: Use a hash like SHA-1
- Signature Creation:
  - Compute the hash of each submitted .dat file
- Audit:
  - Compute the hash of all files on the system
  - Do they match any known signature? If so, alarm!

Signatures based on file hashing

- False positives? Only *very* rarely.
  - For a file f to generate a false alarm, we need

    SHA1(f) == SHA1(x)

    for some malicious file x

Signatures based on file hashing

- False negatives?

Signatures based on file hashing

- False negatives?
  - Technically, none, since
    h = SHA1(x)
    is deterministic given x
  - However…

Attacking File-Hash Signatures

- How could an attacker defeat this approach?

Attacking File-Hash Signatures

- How could an attacker defeat this approach?

- Simple technique:
  - Change the answers to the questions
  - Bob\n40\nOrange → Bob\n41\nOrange

Better Idea: Signatures for Attack Code

- Idea: Use string matching to identify attack code, regardless of its surroundings
- Signature generation:
  - Find the attack code in each malicious sample
  - Save it as a string of bytes
- Audit:
  - Run a string matching algorithm on all files
  - If any file matches any attack code, alarm!

Signatures for Attack Code

- Advantages
  - Doesn’t depend on context – detects the attack code in any file
  - Low (zero?) false alarm rate
- Disadvantages
  - Requires manual effort to find and extract bad code
  - Naïve string matching is extremely slow.
  - Sophisticated string matching is still slow.
  - Number of signatures grows **large**

Better Signatures

- Idea:
  - Don't do string matching
  - Use regular expressions or some other flexible pattern
- Benefits:
  - More concise signature database
- Challenges:
  - Increased false positives

Defeating Signature-Based Detection

- Polymorphic Code
  - There are multiple ways to encode the same series of computations
- Variants
  - Metamorphic code
  - Packers
  - Emulators

Polymorphic Code

- Goal: Defeat signature detection
- Approach: Use different sequences of instructions to achieve the same outcome
- Example: We've already done it. ☺

Polymorphic Code: Example

	jmp <call_instr>	
payload:	movb $0xb, %al	
	popl %ebx	
		xorl %ecx, %ecx
	~~movb $0, 7(%ebx)~~	movb %ecx, 7(%ebx)
	movl %ebx, 0x8(%ebx)	
	~~movl $0, 0xc(%ebx)~~	movl %ecx, 0xc(%ebx)
	leal $0x8(%ebx), %ecx	
	leal $0xc(%ebx), %edx	
	int $0x80	
	movl $1, %eax	
	movl $0, %ebx	
	int $0x80	
	call <payload>	
	.string "/bin/sh"	
	<address of string "/bin/sh" will be written here>	
	<null word will be written here>	

We did this in Week 3!

"Smashing the Stack" shows how to morph our basic shellcode to remove null bytes

Attackers can make similar changes to defeat signature detection!

Polymorphic Code for Homework 2

- Change the length of the NOP sled
  - Defeats string matching, but not regex
- Change the attack payload
- Change the return address

Metamorphic Code

- Metamorphic code transforms **itself** when it reproduces
  – Kind of like a randomized compiler

- Example:
  – Old instruction: x = 2
  – New possibilities:
    - x = 6 – 4;
    - x = 32 / 16;
    - x = 0; x++; x++;
    - x = INT_MAX; x = x + 3;

Packing and Unpacking

- Idea:
  - Store code on disk in a non-standard format
  - Use some additional “stub” code to load and run
- Examples:
  - Self-extracting zip archives
  - UPX http://upx.sourceforge.net/
  - Many more: See http://en.wikipedia.org/wiki/Executable_compression

A Very Simple Unpacker

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<compression.h>

int main(int argc, char* argv[]) {
  void (*fp)(void); // function pointer
  char compressed = [cd366e5514b6b...]; // compressed code
  char *code = malloc(1024*1024);

  decompress(code, compressed);
  mprotect(code, 1024*1024, PROT_EXEC);
  fp = code;
  fp();

  return 0;
}
```

Extract the code

Make pages executable, even though they're on the heap

Run it!

Packing and Unpacking: Strengths

- Generic – Doesn't depend on the packed code
- Packing and unpacking routines can do anything
  – Makes files very difficult to analyze statically
- Packing is not incriminating by itself
  – Lots of commercial software is packed

Packing and Unpacking: Weaknesses

- Defenders can detect the unpacker code
  - Limits re-usability
  - Unpacker code needs to be polymorphic / metamorphic
- A/V companies can easily unpack known packers

Emulator Malware

- Idea:
  - Define a custom byte code “instruction set”
  - Build an interpreter for this instruction set
  - Ship the interpreter, together with malicious code written in the new instruction set

- Challenges:
  - Interpreter still needs to be polymorphic

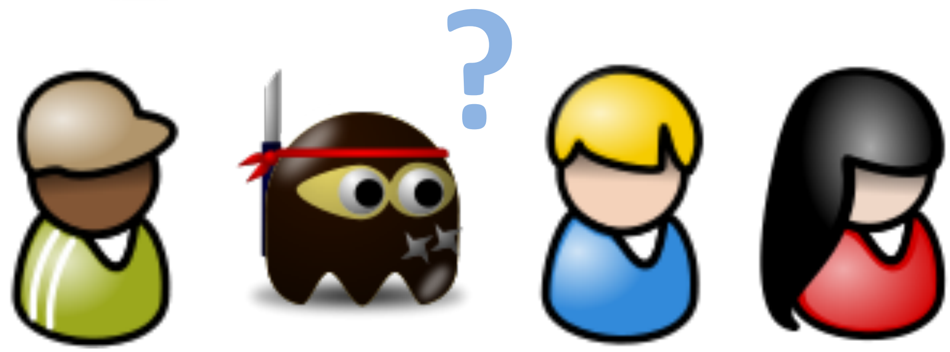
Real-World Data

- See slides from AV Test presentation
  - *Useful and Useless Statistics About Viruses and Anti-Virus Programs*
    by Maik Morgenstern and Hendrik Pilz
    Presented at CARO 2010 Helsinki
    http://www.av-test.org/en/publications/
- Conclusions are not pretty…

Approaches for Audit

- Blacklisting

- Anomaly Detection



- Whitelisting

Anomaly Detection

- Idea: Be suspicious of anything outside the ordinary
- Intuition: Like an *immune system* for the computer
  - Recognizes self vs non-self

Anomaly Detection: Unix Processes

- **A Sense of Self for Unix Processes**
  by Stephanie Forrest et al
  in IEEE Symposium on Security & Privacy, 1996

- Idea:
  - Legitimate Unix programs interact with the system in some predictable way
  - A compromised program will behave differently
  - Use sequences of system calls to capture program behavior

Problem: Mimicry Attacks

- **Mimicry attacks on host-based intrusion detection systems**
  by David Wagner and Paul Soto
  in Proceedings of ACM CCS 2002

- Idea: Make malicious code **act like** good code

Mimicry Attack

- Approach
  - Take normal attack code
  - Insert meaningless “no op” system calls to make it look more like the victim **to the monitor**
- Why it works
  - Monitor doesn’t check system call arguments
  - Attacker can put anything he wants

Example No-Op Syscalls

- fd = open("/tmp/file.txt");
  … // do other things, don't write to fd
  close(fd);

Anomaly Detection: Shellcode

- Idea:
  - English text is all ASCII (byte values < 0x80)
  - Attack code contains all kinds of crazy unprintable characters (byte values >= 0x80)
  - If we see lots of non-ASCII bytes, alarm!

Polymorphic Printable ASCII Shellcode

- See Sections 0x680 and 0x690 in Erikson
- Example: NOP sleds are easy to detect
  - NOP = 0x90 > 0x80 – not printable
  - Long sleds of 0x90 are obvious signs of stack smashing

Polymorphic Printable ASCII Shellcode

- Idea:
  - Replace non-printable instructions
  - Find sequences of printable instructions that do the same thing

- Example: NOP sled
  - Need 1 or more 1-byte instructions that don't do anything meaningful

Polymorphic Printable ASCII Shellcode

- Fortunately x86 gives us lots of options ☺

Instruction	Hex	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x49	K
dec ecx	0x4A	I

Polymorphic Printable ASCII Shellcode

- Old NOP sled
  - 0x90909090…
- New NOP sled
  - ABCABBDABHBCCAAHBBBHBBAAIIIAACKCCKCK

English Shellcode

- **English Shellcode**
  by Josh Mason, Sam Small, Fabian Monrose, and Greg MacManus
  in Proceedings of ACM CCS, 2009.

- Idea:
  - Carry this craziness to its logical conclusion ☺
  - Generate shellcode that “looks like” valid English

English Shellcode: Approach

- Use a statistical model of English text
- Search for words and phrases that
  - Encode our shellcode
  - Are reasonably likely in English
- It works because so much of x86 is printable!

English Shellcode

STORAGE		
ASCII	**HEX**	**ASSEMBLY**
" ca"	20 63 61	`and 61(%ebx), %ah`
" An"	20 41 6E	`and 6E(%ecx), %al`
" jo"	20 6A 6F	`and 6F(%edx), %ch`

JUMPS		
ASCII	**HEX**	**ASSEMBLY**
p.	70 2E	`jo short $30`
q.	71 2E	`jno short $30`
r.	72 2E	`jb short $30`
s.	73 2E	`jnb short $30`
t.	74 2E	`je short $30`
u.	75 2E	`jnz short $30`
v.	76 2E	`jbe short $30`
w.	77 2E	`ja short $30`
x.	78 2E	`js short $30`
y.	79 2E	`jns short $30`
z.	7A 2E	`jpe short $30`

Figure 2 from Mason et al, CCS 2009

STACK MANIPULATION		
ASCII	**HEX**	**ASSEMBLY**
A	41	`inc %eax`
B	42	`inc %edx`
C	43	`inc %ebx`
D	44	`inc %esp`
E	45	`inc %ebp`
F	46	`inc %esi`
G	47	`inc %edi`
H	48	`dec %eax`
I	49	`dec %ecx`
J	4A	`dec %edx`
K	4B	`dec %ebx`
L	4C	`dec %esp`
M	4D	`dec %ebp`
N	4E	`dec %esi`
O	4F	`dec %edi`
P	50	`push %eax`
Q	51	`push %ecx`
R	52	`push %edx`
S	53	`push %ebx`
T	54	`push %esp`
U	55	`push %ebp`
V	56	`push %esi`
W	57	`push %edi`
X	58	`pop %eax`
Y	59	`pop %ecx`
Z	5A	`pop %edx`
a	61	`popa`

English Shellcode

- Extra bonus instruction
  - ASCII ‘r’ means “skip the next X bytes”
  - So the attacker gets a little more freedom

English Shellcode: Approach

- Build attack payload word-by-word
- Look at the last 4 words we've output
- Use a corpus of English text to answer
  - What are all the possible words that might come next?
- Try encoding next bytes of shellcode using each candidate word

English Shellcode: Example Output

... the result of the collapse of large portions of the three provinces to have a syntax which can be found in the case of Canada and the UK, for the carriage of goods were no doubt first considered by the British, and the government, and the Soviet Union operated on the basis that they were...

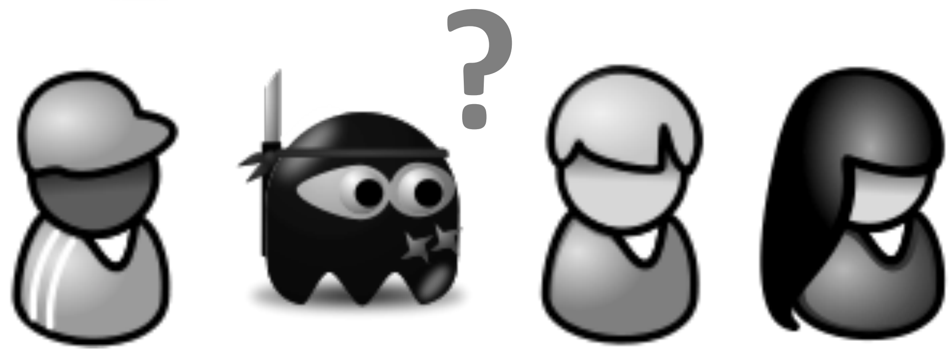
Anomaly Detection: Wrap-Up

- Is it worthwhile?

Approaches for Audit

- Blacklisting

- Anomaly Detection



- Whitelisting

Whitelisting

- Idea: Allow only known good behaviors

Whitelisting Example: Tripwire

- Compute the hash of each important system file

  h = SHA1(file)

- Periodically check whether hashes have changed

  h’ = SHA1(file)

- If a file has changed, raise the alarm!

  h’ == h ?

Tripwire

- False negatives?
  - Would require the adversary to control the OS
- False positives?
  - System files change all the time!
    - Vendor updates
    - On (older) Linux: prelinking
  - How to keep up with them all? Lots of work…
- Also, how do you know your original file was good?

Whitelisting for A/V?

- Idea:
  - Too many different malware variants
    - Can't keep up with them all
  - Relatively few legitimate programs
    - Easier to keep track of these
- Everyone runs Firefox, MS Word, Excel, …
  - Hashes can be computed centrally, then checked everywhere

Whitelisting for A/V?

- Problems: People actually do run lots of programs
  - All big companies have in-house software
  - New open source / freeware every day
  - New versions of old programs (lots of updates…)
  - Software developers – new code every minute

Audit: Conclusions

- There's no silver bullet
  - Defenses can be defeated
  - Or they're expensive and labor-intensive

- But it's not all bad news
  - We have good ways of detecting less sophisticated attacks / attackers
  - These also make the smart attackers' lives harder