Lecture 12: Security

CSC 469 / CSC 2208

Fall 2017
What it’s all about

- Risks: Loss of ...
  - Confidentiality/Privacy, Integrity, Availability/Access
  - Risk Analysis: cost/loss * loss freq. vs. cost to protect
    - Engineering trade-offs, not either-or decisions
    - often, Security == 1 / (functionality * convenience)
  - Vulnerabilities
    - examples abound, many reasons behind
  - Countermeasures
    - carefulness, cryptography, firewalls, detection, recovery
Some key security goals

- Confidentiality
  - keep information content away from the unauthorized
- Integrity
  - prevent undetected, unauthorized modification of data
- Availability
  - ensure that resources and services are available when needed
- Authentication
  - prove the identity of entities or source of information
- Non-repudiation
  - prevent denial of previous commitments
- Privacy and Anonymity
  - prevent Big Brother from invading your space
Some key security problems

- 1. Misplaced trust
- 2. Buggy implementations
- 3. Poor configuration choices
- ...
- 12. Unsafe design assumptions
- ...
- 997. Cryptanalysis
Terminology: Threats

- **Threat**: A potential vector (means, mechanism) for a system’s security to be compromised
  - An *attack* exercises a threat
  - A successful attack leads to a security *compromise*

- Examples of threats:
  - Network traffic arriving from the internet
  - Self-administered systems connected to a corporate (or university) LAN
Terminology: Vulnerabilities

• A *vulnerability* is a flaw in a system that has a security implication

• Examples:
  • Unchecked string copy allows buffer overflow
  • Administrator forgets to disable debug mode on a program during configuration, leaving unsafe but convenient features in deployed service
  • Naïve home user buys wireless router, but does not alter default password on router
  • Compromises occur when attacker matches threats with vulnerabilities
Scary aspects of “bad guys”

• Patience and time
  • Historically successful crackers have been willing to spend endless hours trying to get into systems

• Automated Tools
  • Crackers don’t even have to know anything anymore
    • Copious “cookbooks” and packaged kits
  • One clever person finds a hole, everyone runs her tools

• New profit motive
  • Rent-a-bot-net brokers
IRC Hacker Market

2006 data

Figure 15: Price schedule for compromised hosts.

IRC Hacker Market

...and if that weren’t bad enough

- attackers only need one weakness
  - no need to break thru strongest wall
  - they’ll try lots and exploit the weakest
Example: The Morris Worm

- Released on November 2, 1988
- Written by Robert T. Morris
- Invaded around 6,000 computers within hours (10% of the Internet at the time)
- Disabled many systems and services
- Morris had a friend post instructions for disabling the worm - but it was too late
- Damage estimates between $10,000 and $97 million (shows how hard it is to estimate)
- Details in June 1989 *Comm. of the ACM*
  - “Crisis and Aftermath”, Eugene H. Spafford
How the worm worked

- Copied itself to remote systems via 3 holes
- Exploit hole in finger daemon that caused buffer overflow to create remote shell
  - gets() used to read input
- Exploit hole in Unix sendmail daemon
  - listen()’s on TCP port, accept()’s connections from mailers
  - Exchanges messages about mail envelope and content
  - when running in debug mode, worm could give it commands to execute
  - sendmail ran the malicious code
- Password cracking with a dictionary of 432 words
  - accounts tested against words in a random order
“People pick bad passwords, and either forget, write down, or resent good ones.”

*Steven M. Bellovin*
Effect of worm

- Formation of CERT
- $10,000 fine, 3 year probation, and 400 hours of community service for Morris
- Heightened awareness of computer system vulnerabilities
- Something for security professionals to quote
  - not so much a problem now 😞

<table>
<thead>
<tr>
<th>Notice</th>
<th>Mozilla Releases Multiple Updates</th>
<th>Nov 19, 2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notice</td>
<td>Google Releases Google Chrome 31.0.1650.57</td>
<td>Nov 18, 2013</td>
</tr>
<tr>
<td>TA13-317A</td>
<td>Microsoft Updates for Multiple Vulnerabilities</td>
<td>Nov 13, 2013</td>
</tr>
<tr>
<td>TA13-169A</td>
<td>Oracle Releases Updates for Javadoc and Other Java SE Vulnnerabilities</td>
<td>Jun 18, 2013</td>
</tr>
</tbody>
</table>
Example: ILOVEYOU (2000)

- Email worm – emails had attachment “LOVE-LETTER-FOR-YOU.TXT.vbs”
- Visual basic script, extension hidden by default
  - Copied itself several times and hid the copies in several folders on the victim's hard drive.
  - Added new files to the victim's registry keys.
  - Replaced several different kinds of files with copies of itself.
  - Sent itself through Internet Relay Chat clients as well as e-mail (first 50 addresses in Windows address book)
  - Downloaded a file called WIN-BUGSFIX.EXE from the Internet and executed it.
    - password-stealing application that e-mailed secret information to the hacker’s e-mail address.
ILOVEYOU

- Considered one of the most damaging worms ever
  - Computer Economics (research firm) estimated total costs to clean up exceeded $5 billion
- Disrupted businesses world-wide (more dependence on email by 2000)
  - A year earlier, Melissa got a lot of attention but did little real damage
- Originated in the Philippines
  - Made it clear that this was a global problem
- Success due to social engineering, bad defaults (scripting enabled), errors in security policies for attachments…
Example: Stuxnet (2010)

- Worm that propagates via USB sticks on Windows PCs
  - Actual target is particular model of Siemens PLC
    - Used in many embedded industrial control systems
  - Exploits multiple vulnerabilities, including four “zero-day” exploits
  - Looks for Siemens SIMATIC WinCC/Step 7 controller software
    - If found, infects controller software (using previously unknown, unpatched vulnerability)
    - Read and alters bits of data in the controlled PLC’s
    - Unknown what the intended effect is, but appears very specific
  - Unusually complex, costly to develop, unknown target
    - Lots of speculation!
Example: CryptoLocker (2013)

- Propagates via email attachments

- Once activated:
  - Encrypts data files on the infected computer using strong RSA public-key cryptography
  - Sends a ransom note demanding the computer owner pay for the private decryption key via Bitcoin
  - Attempts to delete Windows Shadow Copy backups before encrypting files
  - Offline backup or online pay-up…
Example: Heartbleed (2014)

- Vulnerability in OpenSSL cryptographic software library
- Bug is in the OpenSSL's implementation of the TLS/DTLS (transport layer security protocols) heartbeat extension
  - improper input validation (due to a missing bounds check)
- Leads to the leak of memory contents from the server to the client and from the client to the server
  - In particular, leaks private keys
Buggy Code

• 85% of CERT Advisories describe problems that cannot be fixed with cryptography.

• Most of these are bugs in code

• But writing correct code is the oldest -- and probably the most difficult -- problem in computer science. We’re not going to solve it any time soon -- and possibly not ever.
Preventing Bugs

• Structuring code properly can help
  • isolate the security-critical sections
  • use better languages and libraries (and programmers)

• Reducing code complexity
  • when was the last time a vendor *deleted* features when shipping a new release?
  • when did you ever see an ad bragging that some Web browser *doesn’t have* Javascript?
  • people don’t understand how to use what’s already there -- so vendors add even more complexity to help people find the knobs and buttons...
Buffer Overruns

- 30-40% of newly-reported holes are due to these
- C uses character arrays for strings (and for arrays)
  - C compilers do no bounds checking on arrays
  - language design makes such automated checking hard
  - programmers often fail to check all lengths and indices
  - common libc functions (e.g., gets, sprintf) also flawed
    - “Safe” versions also flawed
- Too many programmers say “this array is big enough” -- and it often is, for normal purposes…
Stack Overruns

• A particularly nasty form of buffer overrun
  • normal buffer overruns allow corruption of data
  • stack overruns can give control of execution

• Happens when buffers allocated on stack
  • e.g., local procedure variables

• Since return address also on stack…
  • attacker can subvert return address
  • … and insert code to be executed
  • … and point the return address at the new code

• Example: fingerd bug from Morris Worm
Stack Attack

Earlier Frames

Caller’s Frame

sloppy’s Frame

Increasing Addresses

sloppy() {
    int i;
    char In[4];
    printf("Continue (Y/N) ?\n");
    gets(In);
    if (In[0] == ‘N’)
        return;
    /* more code... */
}

exploit address

exploit code (exec sh)

anything

x

x

x

x

x
Format String Bugs

- C library formatted I/O strings can allow almost arbitrary inspection and modification of program if misused
  - First came to light ~2000
    - First known exploit for wu-ftpdp program, June 2000
  - E.g. “printf(inputstr)” where inputstr is supplied by user
    - inputstr can contain format characters (like %x)
    - printf assumes extra arguments have been pushed on stack, dumps stack contents to output
    - %n format allows write to a memory location
    - With other printf options, allows modification of almost any memory location with any value
Format String Attack

Earlier Frames

Caller’s Frame

In
&In
Saved registers
Return address

sloppy2’s Frame

printf’s Frame

Increasing Addresses

sloppy2() {
char In[12];
...
fgets(In, 12, stdin);
printf(In);
...}

“%x %x %x %x %x”

Prints stack contents
Incomplete input checks

• Before using an input value, check it
  • such a simple rule, yet it is so often not done
  • Checking can be harder than it seems

• Examples:
  • Format string vulnerabilities
  • MySpace Samy virus
  • passing NULL, -1, 0, 0xdeadbeaf, etc…
  • Ballista results: 15-35% of bad POSIX parameters cause robustness failures

• Why?
  • Laziness, optimization, forgetsies, reorgs/changes, assumptions, mindsets
From bug to vulnerability

- So you have a buggy user-level application
  - Why is this so bad?
- In general, compromising a process allows attacker to obtain privileges of that process for arbitrary activities
  - Bad for you, but not necessarily bad for the system
  - Compromising a process with root privilege (on Unix systems) provides a lot of power
    - Read/write any file
    - Read/write kernel memory though /dev/kmem
    - Attach to and trace any running process
    - Install kernel modules / change system configuration
Insufficient Domain Separation

- Authorization domains should be clearly separated
  - otherwise, less-privileged code can get more-privileged code to do bad things
- Unfortunately, this is often not the case
- Examples:
  - environment inheritance by setuid programs in UNIX
    - e.g., max file length or number of files open
Security policies are critical

- Most organizations have a stated policy about control of private information and access to resources
- These policies can help guide protocol implementation
  - and can help with political and “clueless user” problems
- If you can’t say what’s important, how am I supposed to protect it?
  - … and why should I bother trying?
Useful “principles” for security

• Principle of Easiest Penetration
  • passwords: crack, sniffing, trojan horses, social engineering

• Principle of Adequate Protection
  • See “risk analysis” and “cost/benefit”…

• Principle of Effectiveness
  • Countermeasures must be used to be useful!!
  • Remember: users are impatient, lazy …
Firewalls

- Perimeter defenses for nets with unsafe hosts
  - we seem to be unable to harden hosts
  - firewalls are an admission that hosts may not be secure
  - firewalls provide a barrier between *us* and *them*

- Single point of control and expertise
  - access limitation and auditing
  - limits communication to/from the outside world
  - only leave a very few machines exposed to direct attack
    - … and minimize their functionality

- Sounds great – how come we’re not done?
OS Security Mechanisms

- Access Controls
- FreeBSD Jails
- Flask Security Architecture (SELinux)
Access Control

• Common Assumption:
  • System knows identity of user (authentication)
  • Access requests pass through some gatekeeper (authorization)

• Implemented using Access Control Matrix
  • Access control list
  • Capability

• Two main types
  • Discretionary Access Control (DAC)
    • User sets access rights for objects they own
  • Mandatory Access Control (MAC)
    • System sets rights that users can’t override
FreeBSD Jails

• Goal: isolation of processes to contain possible damage \textit{without} lots of extra security management complexity

• Built on chroot concept
  • Give process (and all its children) separate view of file system tree (chdir /tmp/limited_fs; chroot /tmp/limited_fs)
  • Originally introduced for development

• Added new “jail” command
  • Each jail has own superuser
  • Privileges of superuser restricted to only affect things inside jail
  • Process in jail isolated from ones outside jail
Flask Security Architecture: Motivation

• No single definition of security suffices
• Need for many policies and even types of policies
• Computer security solutions must be flexible enough to support wide range of security policies
• This policy flexibility must be supported by the OS mechanisms
Defining Policy Flexibility

- Can’t define through a list of known policies
- Defined in context of a state machine model
  - atomic operations to transition from one state to next
  - policy can interpose atomically on set of controlled operations
  - policy may use knowledge of portion of system state
- 3 Requirements of Policy Flexibility
  - Support fine-grained access controls on low-level objects
  - Propagate access rights according to security policy
  - Deal with changes in policy over time, including revoking previously granted permissions
Policy Changes

- Even simplest policies undergo changes
- Risk of enforcing obsolete policy
- Need for effective atomicity in policy changes
- Complicated by migrated permissions
  - access rights explicitly cached in data structures
  - access rights implicitly cached by operations in progress
Popular Mechanisms are Insufficient

- Capability-based systems
  - propagation of access rights
  - Hydra, KeyKOS, EROS: provide enhancements to limit propagation, but still lacking in support for policies
  - SCAP, ICAP, TMach: do not define mechanisms by which policy is queried to validate capabilities
- Interposition
  - mismatch between functional interface and security needs
  - does not support revocation of migrated permissions
Flask Architecture

- Security server
  - provides labeling, access and polyinstantiation decisions
  - security contexts and security identifiers (SIDs)
- Object managers
  - bind labels to objects
  - enforce access decisions
  - direct clients to appropriate instances
- Access Vector Cache (AVC) library
  - coordinates access decisions
  - minimizes performance impact
- Underlying IPC mechanism
  - must provide identification of clients and servers
Architecture Diagram
Revocation Support

• Object manager atomicity requirements
  • Completeness
  • Promptness

• Protocol between security server and object managers
  • system-wide atomicity for policy changes
  • Security server notifies object manager AVC modules
  • AVC modules update cache state, invoke callbacks
  • Callbacks update migrated permissions
  • AVC modules notify security server of completion
Evaluation of Flexibility

- Support for policy changes
  - architecture provides support for system-wide atomicity
  - microkernel meets object manager atomicity requirements
  - other object managers lack support for migrated permissions
- Set of operations controlled by policy
  - fine-grained controls over all object services
- Set of operations that may be invoked by policy
  - object manager interfaces, AVC module interface
- System state available to policy
  - SID pairs sufficient for most policies (DTOS)
  - use prototype to research need for richer interface
Current Status

- SELinux – Security Enhanced Linux
  - Version of Linux created by the NSA and Secure Computing Corporation (SCC)
  - Incorporates University of Utah Flask security model
    - Supports mandatory access control to all objects
    - Separates Object Managers from Security Server
    - Supports various policy configurations
  - Often criticized for being too complicated

- TrustedBSD
  - Part of this system is a port of SELinux extensions to FreeBSD
  - TrustedDarwin is a port of TrustedBSD to the Darwin system
  - Some components of TrustedBSD have spilled over into OS X
    - not sure if this includes Flask implementation
seccomp

- Sandbox mechanism available in Linux

- Performs *filtering* on system calls
  - Filters are written in Berkeley Packet Filter (BPF) language
  - Attached by a process to itself via `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, filter)`;
  - Inherited by children
  - Filter code can access system call number, arguments
    - Allows or denies the system call

Uses?