6.858 Lecture 1

Administrivia:

Lab 1 out today: buffer overflows. Start early. Next week, there will be tutorials on how to get started with lab 1, during office hours.

Introduction:

What is security?

- Achieving some goal in the presence of an adversary.

Many systems are connected to the Internet, which has adversaries. Thus, design of many systems might need to address security, i.e. will the system work when there's an adversary?

High-level plan for thinking about security:

- Policy: the goal you want to achieve. e.g. only Alice should read file F.
- Common goals: confidentiality, integrity, availability.
- Threat model: assumptions about what the attacker could do. e.g. can guess passwords, cannot physically grab file server. Better to err on the side of assuming attacker can do something.
- Mechanism: knobs that your system provides to help uphold policy. e.g. user accounts, passwords, file permissions, encryption.
- Resulting goal: no way for adversary within threat model to violate policy. Note that goal has nothing to say about mechanism.

as long as the threat model is correct, we can satisfy our policies using the implemented mechanisms

Why is security hard? Negative goal.

- Need to guarantee policy, assuming the threat model.
- Difficult to think of all possible ways that attacker might break in. should have thought about that
- Realistic threat models are open-ended (almost negative models).
- Contrast: easy to check whether a positive goal is upheld, e.g. Alice can actually read file F.
- Weakest link matters.
- Iterative process: design, update threat model as necessary, etc.

the threat model may be flawed, the mechanism "software" may have bugs that need patching, there is no 100% secure system

What's the point if we can't achieve perfect security? In this class, we'll push the boundary of each system to see when it breaks.

- Each system will likely have some breaking point leading to compromise.
- Doesn't necessarily mean the system is not useful: depends on context.
- Important to understand what a system can do, and what a system cannot.

In reality, must manage security risk vs. benefit.

- More secure systems means less risk (or consequence) of some compromises.

- Insecure system may require manual auditing to check for attacks, etc.
- Higher cost of attack means more adversaries will be deterred.

Better security often makes new functionality practical and safe. Suppose you want to run some application on your system. Large companies sometimes prohibit users from installing software that hasn't been approved on their desktops, partly due to security. Javascript in the browser is isolated, making it ok (for the most part) to run new code/applications without manual inspection/approval (or virtual machines, or Native Client, or better OS isolation mechanisms). Similarly, VPNs make it practical to mitigate risk of allowing employees to connect to a corporate network from anywhere on the Internet.

What goes wrong #1: problems with the policy.

Example: Sarah Palin's email account.
http://en.wikipedia.org/wiki/Sarah_Palin_email_hack

currently not recommended

- Yahoo email accounts have a username, password, and security questions.
- User can log in by supplying username and password.
- If user forgets password, can reset by answering security Qs.
- Security questions can sometimes be easier to guess than password.
- Some adversary guessed Sarah Palin's high school, birthday, etc.
- Policy amounts to: can log in with either password or security Qs. (no way to enforce "Only if user forgets password, then ...")

this weakens the security posture as now the policy is: people can log in if they know EITHER the password OR the questions

Example: Mat Honan's accounts at Amazon, Apple, Google, etc.
http://www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking/all/

- Gmail password reset: send a verification link to a backup email address.
  - Google helpfully prints part of the backup email address.
  - Mat Honan's backup address was his Apple @me.com account.
- Apple password reset: need billing address, last 4 digits of credit card.
  - Address can be easy, but how to get 4 digits of user's credit card number?
- Amazon: can add a credit card to an account, no password required.
  - Amazon password reset: provide any one of user's credit card numbers.
  - Amazon: will not print credit card numbers. But will print last 4 digits!

try not to rely heavely on what other systems will do that can affect your system security, be cautious

Example: Twitter's @N account hijacking.
https://medium.com/p/24eb09e026dd

- Can be hard for legitimate user to prove they own an account!

poor authentication managment

How to solve?

- Think hard about implications of policy statements.
- Some policy checking tools can help, but need a way to specify what's bad.

- Difficult in distributed systems: don't know what everyone is doing.

What goes wrong #2: problems with threat model / assumptions.

don't make strong assumptions about how people behave, keep in mind that not every one have security/ technical awarness, even the developers and system admins make mistakes that lead to breaches

Example: human factors not accounted for, ex. Phishing attacks.

- User gets email asking to renew email account, transfer money, or ...
- Tech support gets call from convincing-sounding user to reset password.
- "Rubberhose cryptanalysis".

threat models change over time

Example: computational assumptions change over time.

- MIT's Kerberos system used 56-bit DES keys, since mid-1980's.
- At the time, seemed fine to assume adversary can't check all 2^56 keys.
- No longer reasonable: now costs about $100.
- https://www.cloudcracker.com/dictionaries.html
- Several years ago, 6.858 final project showed you can get any key in a day.

DES is not considered secure nowadays

Example: all SSL certificate CAs are fully trusted.

- To connect to an SSL-enabled web site, web browser verifies certificate.
  - Certificate is a combination of server's host name and cryptographic key, signed by some trusted certificate authority (CA).
- Long list (hundreds) of certificate authorities trusted by most browsers.
- If any CA is compromised, adversary can intercept SSL connections with a "fake" certificate for any server host name.
- In 2011, two CAs were compromised, issued fake certs for many domains (google, yahoo, tor, ...), apparently used in Iran (?).
  - http://en.wikipedia.org/wiki/DigiNotar
  - http://en.wikipedia.org/wiki/Comodo_Group
- In 2012, a CA inadvertently issued a root certificate valid for any domain.
  - http://www.h-online.com/security/news/item/Trustwave-issued-a-man-in-the-middle-certificate-1429982.html

Example: assuming your hardware is trustworthy.

- If NSA is your adversary, turns out to not be a good assumption.
  - https://www.schneier.com/blog/archives/2013/12/more_about_the.html

Example: assuming good randomness for cryptography.

- Need high-quality randomness to generate the keys that can't be guessed.
  - Problem: embedded devices, virtual machines may not have much randomness.
  - As a result, many keys are similar or susceptible to guessing attacks.
- https://factorable.net/weakkeys12.extended.pdf

Example: subverting military OS security.

- In the 80's, military encouraged research into secure OS'es.

- One unexpected way in which OS'es were compromised: adversary gained access to development systems, modified OS code.

physical hacking and red teaming

Example: subverting firewalls.

- Adversaries can connect to an unsecured wireless behind firewall
- Adversaries can trick user behind firewall to disable firewall
- Might suffice just to click on link http://firewall/?action=disable
  - Or maybe buy an ad on CNN.com pointing to that URL (effectively)?

Example: machines disconnected from the Internet are secure?

- Stuxnet worm spread via specially-constructed files on USB drives.

How to solve?

- More explicit threat models, to understand possible weaknesses.
- Simpler, more general threat models.
- Better designs may eliminate / lessen reliance on certain assumptions.
  - E.g., alternative trust models that don't have fully-trusted CAs.
  - E.g., authentication mechanisms that aren't susceptible to phishing.

What goes wrong #3: problems with the mechanism -- bugs.

Bugs in the security mechanism (e.g., OS kernel) lead to vulnerabilities. If application is enforcing security, app-level bugs lead to vulnerabilities.

Example: Apple's iCloud password-guessing rate limits.
https://github.com/hackappcom/ibrute

- People often pick weak passwords; can often guess w/ few attempts (1K-1M).
- Most services, including Apple's iCloud, rate-limit login attempts.
- Apple's iCloud service has many APIs.
- One API (the "Find my iPhone" service) forgot to implement rate-limiting.
- Adversary could make many attempts at guessing passwords.
- Probably as fast as they can send packets: >>M/day.

mechanisms where not equally enforced

Example: Missing access control checks in Citigroup's credit card web site.
http://www.nytimes.com/2011/06/14/technology/14security.html

- Citigroup allowed credit card users to access their accounts online.
  - Login page asks for username and password.
  - If username and password OK, redirected to account info page.
- The URL of the account info page included some numbers.
  - Turns out the numbers were (related to) the user's account number.
  - Worse yet, server didn't check that you were logged into that account.
- Adversary tried different numbers, got different people's account info.
- Possibly a wrong threat model: doesn't match the real world?
  - System is secure if adversary browses the web site through browser.

IDOR

    - System not secure if adversary synthesizes new URLs on their own.
- Hard to say if developers had wrong threat model, or buggy mechanism.

Example: Android's Java SecureRandom weakness leads to Bitcoin theft.
https://bitcoin.org/en/alert/2013-08-11-android

- Bitcoins can be spent by anyone that knows the owner's private key.
- Many Bitcoin wallet apps on Android used Java's SecureRandom API.
- Turns out the system was sometimes forgetting to seed the PRNG!
- As a result, some Bitcoin keys turned out to be easy to guess.
- Adversaries searched for guessable keys, spent any corresponding bitcoins.

all set to 0

Example: bugs in sandbox (NaCl, Javascript, Java runtime).

- Allows adversary to escape isolation, do operations they weren't supposed to.

Example: Moxie's SSL certificate name checking bug

- Null byte vs. length-encoding.

Example: buffer overflows (see below).

Case study: buffer overflows.
Consider a web server. Often times, the web server's code is responsible for security. E.g., checking which URLs can be accessed, checking SSL client certs, etc. Thus, bugs in the server's code can lead to security compromises.

thinking about a web server in context of policies and threat models is by having the idea that the policy is "the server should do what the programmer intended not what the code exactly says"

What's the threat model, policy?

- Assume that adversary can connect to web server, supply any inputs.
- Policy is a bit fuzzy: only perform operations intended by programmer?
- E.g., don't want adversary to steal data, bypass checks, install backdoors.

Consider the following simplified example code from, say, a web server:

parse buf into an integer value using the atoi() function, and stores it in the variable i

```
int read_req(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return i;
}
```

reads a line of text from the standard input and stores it in the character array buf. Note that the gets() function is generally considered unsafe, as it can cause buffer overflows if the input is too long. It's better to use fgets() instead, which takes a maximum length parameter.

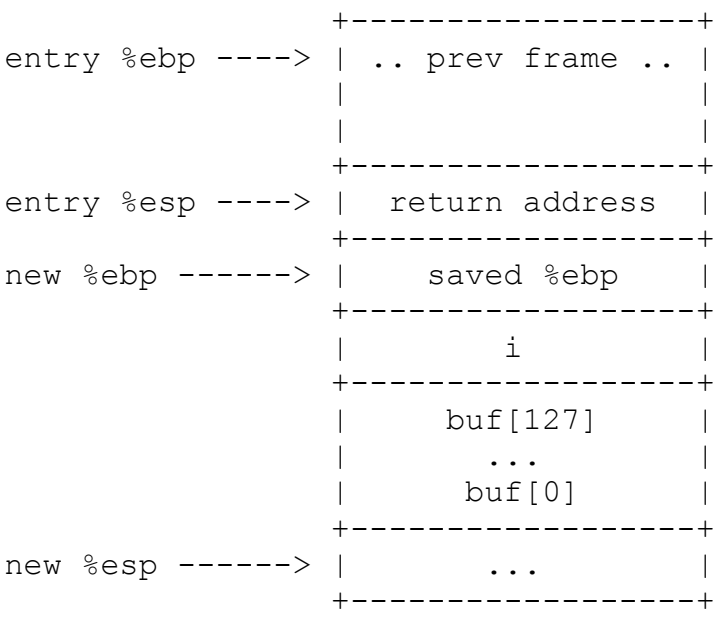
this function, read_req(), reads a line of text from the user, converts it to an integer, and returns that integer value.

Demo to go along with the discussion below:

```
% make
% ./readreq
1234
% ./readreq
12341234123412341234
% ./readreq
AAAAAAAAAAAA....AAAA
% gdb ./readreq
b read_req
r
disas $eip
info reg
print &buf[0]
x $ebp
x $ebp+4
disas 0x08048e5f
next

AAAAAAA...AAA
print &buf[0]
x $ebp
x $ebp+4
next

print &buf[0]   ## why just 128 bytes now?
x $ebp
x $ebp+4

disas $eip
nexti
nexti
disas $eip
info reg
x $esp
stepi
stepi

..
disas main
set {int}$esp = 0x..  ##from main
c
```

What does the compiler generate in terms of memory layout?
x86 stack:

- Stack grows down.
- %esp points to the last (bottom-most) valid thing on the stack.
- %ebp points to the caller's %esp value.



Caller's code (say, main):

```
call read_req
```

read_req's code:

```
push    %ebp
mov     %esp -> %ebp
sub     168, %esp        # stack vars, etc
...
mov     %ebp -> %esp
pop     %ebp
ret
```

How does the adversary take advantage of this code?

- Supply long input, overwrite data on stack past buffer.
- Interesting bit of data: return address, gets used by 'ret'.
- Can set return address to the buffer itself, include some code in there.

How does the adversary know the address of the buffer?

- What if one machine has twice as much memory?
- Luckily for adversary, virtual memory makes things more deterministic.
- For a given OS and program, addresses will often be the same.

What happens if stack grows up, instead of down?

- Look at the stack frame for gets.

What can the adversary do once they are executing code?

- Use any privileges of the process.
- Often leverage overflow to gain easier access into system.
  - Originally on Unix, run shell /bin/sh (thus, "shell code").
- If the process is running as root or Administrator, can do anything.
- Even if not, can still send spam, read files (web server, database), etc.
- Can attack other machines behind a firewall.

Why would programmers write such code?

- Legacy code, wasn't exposed to the internet.
- Programmers were not thinking about security.
- Many standard functions used to be unsafe (strcpy, gets, sprintf).
- Even safe versions have gotchas (strncpy does not null-terminate).

More generally, any memory errors can translate into a vulnerability.

- Using memory after it has been deallocated (use-after-free).
  - If writing, overwrite new data structure, e.g. function ptr.
  - If reading, might call a corrupted function pointer.
- Freeing the same memory twice (double-free).
  - Might cause malloc to later return the same memory twice.
- Decrementing the stack ptr past the end of stack, into some other memory.
  - http://www.invisiblethingslab.com/resources/misc-2010/xorg-large-memory-attacks.pdf
- A one-byte stray write can lead to compromise.
  - http://www.openwall.com/lists/oss-security/2014/08/26/2
- Might not even need to overwrite a return address or function pointer.
  - Can suffice to read sensitive data like an encryption key.
  - Can suffice to change some bits (e.g. int isLoggedIn, int isRoot).

How to avoid mechanism problems?

- Reduce the amount of security-critical code.
  - Don't rely on the entire application to enforce security.
  - Lab 2.
- Avoid bugs in security-critical code.
  - E.g., don't use gets(), use fgets() which can limit buffer length.
  - Use common, well-tested security mechanisms ("Economy of mechanism").
  - Audit these common security mechanisms (lots of incentive to do so).
  - Avoid developing new, one-off mechanisms that may have bugs.
  - Good mechanism supports many uses, policies (more incentive to audit).
- Examples of common mechanisms:

- OS-level access control (but, could often be better)
- network firewalls (but, could often be better)
- cryptography, cryptographic protocols.

6.858 Computer Systems Security

Fall 2014