Writing Secure Code

Lecture 24

A Security Process

• Specify Security Properties
• Design for Security
• Code for Security
• Test for Security

Recommended textbook: "Writing Secure Code"

Design for Security

Common Mistakes in Design for Security

• Common mistake #1
  - Development teams do not understand security
  "We'll add crypto and we'll be done!"

• Common mistake #2:
  - Security is treated as any other feature
  "We can add security later"
  - Security is as strong as the weakest link
  1. You have it when all parts are secure
  2. You can easily add lack-of-security, but not security

Why Are These Mistakes Made?

• Security is a feature disabler
  - But, is leaking your private data a feature?
  - We ought to disable dangerous features

• Security is difficult to measure
  - When do you have more security than competition?
  - Can show when the costs related to attacks drop
Design for Security Principles

• Consider security from the start
• Learn from mistakes (attacks)
• Use least privilege
• Assume external systems are insecure
• Plan on failure
• Use secure defaults

Identify Threat Models Early

• Brainstorm early to predict threats
  - Threat, vulnerability, attacks and motives
• Rank threats
  - Rank: criticality / chance
• Choose how to address threats
  - Authentication, authorization, encryption, ...
• Decide how will you test your protection

Coding For Security

Public Enemy #1: Buffer Overrun

• Danger in unsafe languages, like C or C++
  - Typically designed when machines were weak and performance mattered more than security
• Array accesses are unchecked
  - Checks require additional code

Buffer Overruns

• A buffer overrun writes past the end of an array
• Buffer usually refers to a C array of char
  - But can be any array
• So who's afraid of a buffer overrun?
  - Cause a core dump
  - Can damage data structures
  - What else?

Stack Smashing

Buffer overruns can alter the control flow of your program!

```c
char buffer[100]; /* stack allocated array */
```

![Buffer overrun example diagram]
An Overrun Vulnerability

```c
void foo(char in[]) {
    char buffer[100];
    int i = 0;
    for(i = 0; in[i] != '\0'; i++) {
        buffer[i] = in[i];
    }
    buffer[i] = '\0';
}
```

An Interesting Idea

```c
char in[104] = {' ',…,' ', magic 4 chars}
foo(in); (**)
```

Discussion

- So we can make `foo` jump wherever we like.
- How is this possible?
- Unanticipated interaction of two features:
  - Unchecked array operations
  - Stack-allocated arrays and return addresses
    - Knowledge of frame layout allows prediction of where array and return address are stored
    - Note the "magic cast" from char's to an address

The Rest of the Story

- Say that `foo` is part of a network server and the `in` originates in a received message
  - Some remote user can make `foo` jump anywhere!
- But where is a "useful" place to jump?
  - Idea: Jump to some code that gives you control of the host system (e.g. code that spawns a shell)
  - But where to put such code?
  - Idea: Put the code in the same buffer and jump there!

The Plan

- We'll make the code jump to the following code:
  - In C: `exec("/bin/sh");`
  - In assembly (pretend):
    ```
    mov $a0, 15          ; load the syscall code for "exec"
    mov $a1, &Ldata ; load the command
    syscall ; make the system call
    Ldata: .byte '/','b','i','n','/','s','h',0 ; null-terminated
    ```
  - In machine code: 0x20, 0x42, 0x00, …
- The last 4 bytes in "in" equal the start of buffer
  - A variety of ways to guess it
The State of C Programming

- The most common way to copy the bad code in a stack buffer is using string functions: strcpy, strcat, etc.
- Buffer overruns are common
  - Programmers must do their own bounds checking
  - Easy to forget or be off-by-one or more
  - Program still appears to work correctly
- In C w.r.t. to buffer overruns
  - Easy to do the wrong thing
  - Hard to do the right thing

The State of Hacking

- Buffer overruns are the attack of choice
  - 40-50% of new vulnerabilities are buffer overrun exploits
  - Many recent attacks of this flavor: Code Red, Nimda, MS-SQL server
- Highly automated toolkits available to exploit known buffer overruns
  - Search for "buffer overruns" yields > 25,000 hits

The Sad Reality

- Even well-known buffer overruns are still widely exploited
  - Hard to get people to upgrade millions of vulnerable machines
- We assume that there are many more unknown buffer overrun vulnerabilities
  - At least unknown to the good guys

Do Not Roll Your Own Crypto

- Even though you learn the algorithms in CS70
- Do not try things like
  ```c
  for(i=0;i<n;i++) { b[i] ^= key[i % keylen]; }
  ```
- Can find the key if you have a pair of clear text and its encryption
  - This is sometimes called encryption
- Homegrown crypto may be vulnerable to timing attacks

Cryptographic Keys

- A key is a secret that is used to encrypt and decrypt data
  - The longer it is (more bits), the harder to break
    - DES: 56 bits thought sufficient in 1977. Now 112/128 bits
    - RSA: 1024 bits (equiv. 80 bits DES)
  - Distributed brute force attacks very effective
    - 1997: took 96 days (70,000 users)
    - 1998: 41 days, 1999: 22 hours (250 billion keys/sec)
  - You have to protect info for the future as well!

Choosing Cryptographic Keys

- Best: Choose a random key for a given length
  - Keys are hard to remember
  - Many systems allow you to use a password instead

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Chars</th>
<th>Length(56bits)</th>
<th>Length(128bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>10</td>
<td>17</td>
<td>40</td>
</tr>
<tr>
<td>Alpha (no case)</td>
<td>26</td>
<td>12</td>
<td>28</td>
</tr>
<tr>
<td>Alpha (case)</td>
<td>52</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>AlphaNum</td>
<td>62</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>AlphaNumPunct</td>
<td>93</td>
<td>9</td>
<td>20</td>
</tr>
</tbody>
</table>
Managing Crypto Keys

- DVD key leak (obtained from one software)
- Do not hard code secret keys in code
  - Easy to dump all strings (string command)
  - Easy to look for randomness in code (ncipher.com)
  - Too many people have the same code
- Do not pass keys around (as arguments)
  - A secret that many know is not a secret
  - Pass handles to the key instead (callback to encrypt or decrypt)

Storing Secrets

- Impossible to store secrets perfectly secure

- Attack methods
  - Attach to the process with a debugger
  - Wait for memory to be paged out, read page file
  - Make the application crash, and look at the dump
- Remember: bad guys can install the software on machines the control completely!

Alternatives to Storing Secrets

- Get the secret from user every time it is needed
- If you need to store a secret only to check that somebody else also knows the secret
  - Store a digest in that case
- A digest (or hash, or one way) function
  - A different digest for each input
  - Given the digest, very hard to guess the input
  - Examples: RC5 (by RSA), SHA-1 (by NIST & NSA)
- Keep the digest, then compare it with the digest of the new input

Denial of Service Attacks (DoS)

- Malicious user prevents your application from servicing its legitimate users
- Some of the most difficult forms of attacks to protect against
- People often dismiss these attacks because the malicious user never gets to "do" anything
  - Problem is, nobody else can "do" anything

DoS: Application Failure

- Malicious user makes your application crash
- Example: any memory error can first be exploited to generate a crash
  - It is much harder to get elevated privilege
- Example: ping of death
  - Some OSs crash if they get a UDP packet too long
- Really, a software quality issue

DoS: CPU Starvation

- Force an application to get stuck in a loop computing something, preferably forever
- Example: an application wants to convert all double '/' to single ones in user input
  for(t=buf; *t && *(t+1); t++) {
    if(*t == '/' && *(t+1) == '/') { strcpy(t, t+1); }
  }
- Takes $n^2$ time on a string of $n$ '/'
- Use efficient algorithms for user input
  - Or, decide quickly to reject the input
DoS: Resource Starvation

- Force an application to consume too many resources
  - memory, database connections, opened files, ...
- Example:
  - Accept a connection, allocate resources to handle the connection, then see who is at the other end ...
- Do not allocate expensive resources until you know you are talking to a legitimate user
  - Do not allow an attacker to cause you to do expensive operations

DoS: Quotas

- What if you cannot distinguish the valid users?
  - For a web server, each connection is possibly valid
- One solution: quotas
- Quotas are a source of DoS attacks
  - Somebody hogs resources, and the server stops accepting legitimate connections
- Better: per user or per source address quotas
  - Must be configurable
- Better: quotas vary with the system load

DoS: Network Bandwidth Attacks

- Example: some version of RPC
  - Replies with an error packet to unexpected packets
  - Can lead to a flood of error packets back and forth
- Only reply to packets that conform to protocol
- Do not reply to packets sent to broadcast addresses
- As in real life, some inputs are best ignored

Running with Least Privilege

- Software must run with least privilege compatible with the legitimate needs
  - Even when compromised, can do less damage
- How many of you log in as administrator?
  - Design applications to be used by non-administrator
  - Occasionally, must run with more privilege (sudo)
- Determine least privilege
  - Find all resources needed
  - Find all privileged API calls
  - Test applications as a regular user !!

Never Trust User Input

- All input is bad until proven otherwise
  - Especially true for web-based applications
  - Check input always
  - Never pass user input directly to a shell or interpreter
- Example:
  - Ask user for name and password, then:
    ```
    val strSQL = "SELECT count(*) FROM client WHERE name = " + name + " and pwd=" + pwd;
    if(runSQL(strSQL).Value > 0) ...;
    
    User gives pwd: foo or true
    Or, user gives name: admin -- (-- starts comment)
    ```

Securing Web Applications

- Just like the above, except you can count on malicious users
  - Always check the input
  - Do not do security checks in client side script
  - Do not store sensitive data in cookies or hidden fields
- Cross scripting attacks
  - Somebody leaves a comment that contains HTML tags with scripts
  - Later viewers will run the script
Testing For Security

- Just like with quality, testing never adds security to a product
- Assume bad people have access to the code
  - Do white-box testing
  - A security review is more important
- Test for known vulnerabilities
  - E.g., for cross-site scripting: `<script>alert</script>`

Conclusions

- Design and code with security in mind
  - From the start, at all stages
- Important to know security vulnerabilities
  - Often it takes devious mind to figure them out