Exploiting Software: How to Break Code

Gary McGraw, Ph.D.
CTO, Cigital

http://www.cigital.com

- Software Quality Management solutions
- Security
- Reliability
- Performance
- Fortune 1000 clients
- Cigital Labs
Pop quiz

What do wireless devices, cell phones, PDAs, browsers, operating systems, servers, personal computers, public key infrastructure systems, and firewalls have in common?

Software

So what’s the problem?
Commercial security is reactive

- Defend the perimeter with a firewall
  - To keep stuff out
- Over-rely on crypto
  - “We use SSL”
- “Review” products when they’re done
  - Why your code is bad
- Promulgate “penetrate and patch”
- Disallow advanced technologies
  - Extensible systems (Java and .NET) are dangerous

The “fat guy with keys” does not really understand software development.

Builders versus operators

- Most security people are operations people
  - Network administrators
  - Firewall rules manipulators
  - COTS products glommers
  - These people need training
- Security means different things to different people

- Most builders are not security people
  - Software development remains a black art
  - How well are we doing teaching students to engineer code?
  - Emergent properties like security are hard for builders to grok
  - These people need academic education
Making software behave is hard

- Can you test in quality?
- How do you find (adaptive) bugs in code?
- What about bad guys doing evil on purpose?

- What’s the difference between security testing and functional testing?
- How can you teach security design?
- How can you codify non-functional, emergent requirements like security?
- Can you measure security?

Attaining software security is even harder

**The Trinity of Trouble**

- **Connectivity**
  - The Internet is everywhere and most software is on it

- **Complexity**
  - Networked, distributed, mobile code is hard

- **Extensibility**
  - Systems evolve in unexpected ways and are changed on the fly
Software complexity growth

Windows Complexity

Millions of Lines

Win
Win
Win
Win
Win
Win
XP
3.1
NT
95
98
5.0
2K
NT 4.0
(1990)
(1995)
(1997)
(1998)
(1999)
(2000)
(2001)
(2002)

Software vulnerability growth

Vulnerabilities Reported to CERT/CC

Total vulnerabilities reported (1995-2002): 9,162

© 2004 Cigital
Basic understanding of complexity and its impact on security problems is sorely needed.

Do the LOC and vulnerability graphs correlate?
- Dan Geer’s power law prediction

What are software security problems really like?
- How common are basic categories?
- How can we teach students something that now takes years of fieldwork to merely intuitively grasp?

Who is the bad guy?
- Hackers
  - “Full disclosure” zealots
  - “Script kiddies”
- Criminals
  - Lone guns or organized
- Malicious insiders
  - Compiler wielders
- Business competition
- Police, press, terrorists, intelligence agencies
Modern security is about managing risks

- There is no such thing as 100% secure
- Must make tradeoffs
- Should be business decisions
- Proactive security is about building things right
  - Design for security
  - Security analysis
  - Secure coding practice
  - Security testing

It's all about the software
- Most security problems are caused by software bugs and flaws
- We must build secure software

It's all about the software

Know your enemy: How stuff breaks
Security problems are complicated

**IMPLEMENTATION BUGS**
- Buffer overflow
  - String format
  - One-stage attacks
- Race conditions
  - TOCTOU (time of check to time of use)
- Unsafe environment variables
- Unsafe system calls
  - System()
- Untrusted input problems

**ARCHITECTURAL FLAWS**
- Misuse of cryptography
- Compartmentalization problems in design
- Privileged block protection failure (DoPrivilege())
- Catastrophic security failure (fragility)
- Type safety confusion error
- Insecure auditing
- Broken or illogical access control (RBAC over tiers)
- Method over-riding problems (subclass issues)
- Signing too much code

**BUG: The dreaded buffer overflow**
- Overwriting the bounds of data objects
- Allocate some bytes, but the language doesn’t care if you try to use more
  - char x[12];
    x[12] = ‘\0’;
- Why was this done? Efficiency!
- Two main flavors of buffers
  - Heap allocated buffers
  - Stack allocated buffers
  - Smashing the stack is the most common attack
- The most pervasive security problem today in terms of reported bugs

![Graph showing security problems from 1988 to 1999](image)
Pervasive C problems lead to BUGS

```c
void main() {
    char buf[1024];
    gets(buf);
}
```

- Calls to watch out for:
  - `gets(buf)`
  - `strcpy(dst, src)`
  - `strncpy(dst, src, n)`
  - `strcat(dst, src)`
  - `strncat(dst, src, n)`
  - `sprintf(buf, fmt, a1, …)`
  - `snprintf(buf, fmt, a1, n1, …)`

- How not to get input:
  - Attacker can send an infinite string!
  - Chapter 7 of K&R (page 164)

- Use static analysis to find these problems:
  - ITS4, SourceScope
  - Careful code review is necessary

FLAW: Architectural problems with Java

- Java's classloading architecture broken
  - Separate instantiate class from manage name spaces
  - Every release had problems:
    - March 96: JDK 1.0
    - March 97: JDK 1.0.7
    - July 98: JDK 1.2

- What is “Java” anyway? (and what is .NET?!)

© 2004 Cigital
**Attackers do not distinguish bugs and flaws**

- Both bugs and flaws lead to vulnerabilities that can be exploited
- Attackers write code to break your code
- The standard attacker’s toolkit has lots of stuff
  - Disassemblers and decompilers
  - Control flow and coverage tools
  - APISPY32
  - Breakpoint setters and monitors
  - Buffer overflow
  - Shell code
  - Rootkits

---

**Attacker’s toolkit: disassemblers and decompilers**

- Source code is not a necessity for software exploit
- Binary is just as easy to understand as source code
- Disassemblers and decompilers are essential tools
- Reverse engineering is common and must be understood (not outlawed)
- IDA allows plugins to be created
- Use bulk auditing
Attacker’s toolkit: control flow and coverage

- Tracing input as it flows through software is an excellent method.
- Exploiting differences between versions is also common.
- Code coverage tools help you know where you have gotten in a program.
  - dyninstAPI (Maryland)
  - Figure out how to get to particular system calls
  - Look for data in shared buffers

Attacker’s toolkit: APISPY32

- Look for broken system calls (at all levels in code)
- lstrcpy() makes a great example
- On win32 systems, use APISPY to determine which APIs are being used by a target program
- Interposition attacks are a great thing to think about at this level
Attacker’s toolkit: breakpoints

- Breakpoints are central to debuggers
  - Use interrupt 3 on x86 architectures
  - Mark entire blocks for access
  - Single step at breakpoint (also as in debugging)

- Check out “The PIT” http://www.hbgary.com

Attacker’s toolkit: the buffer overflow

- Find targets with static analysis
- Change program control flow
  - Heap attacks
  - Stack smashing
  - Trampolining
- Particular examples
  - Overflow binary resource files (used against Netscape)
  - Overflow variables and tags (Yamaha MidiPlug)
  - MIME conversion fun (Sendmail)
  - HTTP cookies (apache)

- Trampolining past a canary

<table>
<thead>
<tr>
<th>Function arguments</th>
<th>Return Address</th>
<th>Canary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Pointer</td>
<td>Local Variable: Buffer A</td>
<td></td>
</tr>
<tr>
<td>Local Variable: Pointer A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Variable: Buffer B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Attacker’s toolkit: shell code and other payloads

- Common payloads in buffer overflow attacks
- Size matters (small is critical)
- Avoid zeros
- XOR protection (also simple crypto)

- Payloads for
  - X86 (win32)
  - RISC (MIPS and sparc)
  - Multiplatform payloads

Attacker’s toolkit: rootkits

- The apex of software exploit…complete control of the machine
- Live in the kernel
  - XP kernel rootkit in the book
  - See http://www.rootkit.com
- Get into the microchips (hardware viruses)
- Hide files and directories by controlling access to process tables
- Provide control and access over the network
Attacker’s toolkit: other miscellaneous tools

- Debuggers (user-mode)
- Kernel debuggers
  - SoftIce
- Fault injection tools
  - Failure simulation tool
  - Hailstorm
  - Holodeck
- Boron tagging
- The “depends” tool
- Grammar rewriters

How attacks unfold

- Attacking a system is a process of discovery and exploration
  - Qualify target (focus on input points)
  - Determine what transactions the input points allow
  - Apply relevant attack patterns
  - Cycle through observation loop
  - Find vulnerability
  - Build an exploit

The standard process

- Scan network
- Build a network map
- Pick target system
- Identify OS stack
- Port scan
- Determine target components
- Choose attack patterns
- Leverage environment faults
- Use indirection
- Plant backdoor
Knowledge: 48 Attack Patterns

- Make the Client Invisible
- Target Programs That Write to Privileged OS Resources
- Use a User-Supplied Configuration File to Run Commands That Elevate Privilege
- Make Use of Configuration File Search Paths
- Direct Access to Executable Files
- Embedding Scripts within Scripts
- Leverage Executable Code in Nonexecutable Files
- Argument Injection
- Command Definitions
- Multiple Parse and Double Escapes
- User-Supplied Variable Passed to File System Calls
- Postfix NULL Terminator
- Relative Path Traversal
- Client-Controlled Environment Variables
- User-Supplied Global Variables (DEBUG=1, PHP Globals, and So Forth)
- Session ID, Resource ID, and Blind Trust
- Attack Pattern Fragment: Manipulating Terminal Devices
- Simple Script Injection
- Embedding Script in Nonscript Elements
- XSS in HTTP Headers
- HTTP Query Strings
- User-Controlled Filename
- Passing Local Filenames to Functions That Expect a URL
- Meta-characters in E-mail Header
- File System Function Injection, Content Based
- Client-Side Injection, Buffer Overflow
- Cause Web Server Misclassification
- Alternate Encoding the Leading Ghost Characters
- Using Slashes in Alternate Encoding
- Using Escaped Slashes in Alternate Encoding
- Unicode Encoding
- UTF-8 Encoding
- URL Encoding
- Alternative IP Addresses
- Slashes and URL Encoding Combined
- Web Logs
- Overflow Binary Resource File
- Overflow Variables and Tags
- Overflow Symbolic Links
- MIME Conversion
- HTTP Cookies
- Filter Failure through Buffer Overflow
- Buffer Overflow with Environment Variables
- Buffer Overflow in an API Call
- Buffer Overflow in Local Command-Line Utilities
- Parameter Expansion
- String Format Overflow in syslog

Attack pattern 1: Make the client invisible

- Remove the client from the communications loop and talk directly to the server
- Leverage incorrect trust model (never trust the client)
- Example: hacking browsers that lie
Attack pattern 2: Command delimiters

- Use off-nominal characters to string together multiple commands

- Example: shell command injection with delimiters

```html
<input type=hidden name=filebase value="bleh; [command]">
```

```
exec("cat data_log_"; rm -rf /; cat temp.dat)
```

Attack pattern 3: Cross site scripting

- XSS
  1. Attacker sends active content to a victim
  2. Content invokes a script on the vulnerable website
  3. Later invoked by a web browser hitting the website
  4. The script runs
  5. Attacker allowed access

- Examples
  - Javascript injection
  - Inject in non-script elements
  - HTTP headers
  - Query strings

```html
<script>
evalScript();
</script>
```
Breaking stuff is important

- Learning how to think like an attacker is essential
- Do not shy away from teaching attacks
  - Engineers learn from stories of failure
- Attacking group projects can be the most fun part of a course
- Fun is good! Software engineering is too boring!

Great, now what do we do about this?
Software security critical lessons

- Software security is more than a set of security functions
  - Not magic crypto fairy dust
  - Not silver-bullet security mechanisms
  - Not application of very simple tools
- Non-functional aspects of design are essential
- Security is an emergent property of the entire system (just like quality)
- To end up with secure software, deep integration with the SDLC is necessary

Towards a solution

- Integrate software security best practices into EVERYTHING you build
- Understand and follow guiding security principles
- If you write code, learn about software security!
Ten guiding principles for secure design

1. Secure the weakest link
2. Practice defense in depth
3. Fail securely
4. Follow the principle of least privilege
5. Compartmentalize
6. Keep it simple
7. Promote privacy
8. Remember that hiding secrets is hard
9. Be reluctant to trust
10. Use your community resources

The antidote: Software security in the SDLC

- Security requirements
- External review
- Static analysis (tools)
- Penetration testing
- Risk analysis
- Risk-based security tests
- Risk analysis
- Security breaks

Abuse cases

Requirements and use cases
- Design
- Test plans
- Code
- Test results
- Field feedback
Best practices reprise

- These best practices should be applied throughout the lifecycle
- Tendency is to “start right” (penetration testing) and declare victory
  - Not cost effective
  - Hard to fix problems
- Start as far to the left as possible
- Abuse cases
- Security requirements analysis
- Architectural risk analysis
- Risk analysis at design
- External review
- Test planning based on risks
- Security testing (malicious tests)
- Code review with static analysis tools

Where to learn more
New Department on Software Security best practices called “Building Security In”

http://www.computer.org/security

Cigital’s Software Security Group invents and practices Software Quality Management

http://www.cigital.com/presentations/sploit04

Use Exploiting Software and Building Secure Software

Send e-mail: gem@cigital.com