Secure Computing - Part 1

Deborah Stacey

Computing and Information Science
University of Guelph

July 3, 2007
Outline of Part I

1 Secure Computing
   - References
   - The Problem
   - Vulnerability/Patch/Alarm Cycle
   - Attacks
   - Architectural / Design
   - Implementation Attacks
   - Operations Attacks
   - Factors that work against Secure Code
   - A Case Study: rlogin
Outline of Part II

2. Defensive Programming
   - Defensive Programming
   - Other Sources of Information on Secure Code
   - Top 10 Secure Coding Practices
Part I

Why Secure Computing?
References

  - First chapter is available at http://www.oreilly.com/catalog/securecdng/chapter/index.html

- **CERT**
  - The CERT Coordination Center is a centre of Internet security expertise located at the Software Engineering Institute at Carnegie Mellon University.
  - They handle computer security incidents and vulnerabilities, publishing security alerts, R&D into security, and education
  - http://www.cert.org
Software Attacks

- CERT statistics on identified vulnerabilities
  (http://www.cert.org/stats/

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerabilities</td>
<td>171</td>
<td>345</td>
<td>311</td>
<td>262</td>
<td>417</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>1Q 2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerabilities</td>
<td>1,090</td>
<td>2,437</td>
<td>4,129</td>
<td>3,784</td>
<td>3,780</td>
<td>5,990</td>
<td>8,064</td>
<td>2,176</td>
</tr>
</tbody>
</table>

Total vulnerabilities reported (1995-Q1, 2007): 32,956
Estimates of the Scope of the Problem

- There may be 1 security bug per 1000 LOC.

- An OS can contain 100 million LOC.

- Even if bugs are detected, security patches may inject other bugs.

- It is estimated that 10% to 15% of all fixes introduce new bugs.
Vulnerability/Patch/Alarm Cycle

- Discover Vulnerability
- Develop Patch
- Get Alert
- Install Patches
What is an Attack?

- An attack is any malicious act against a system or systems.

- An attack has
  - Goals / subgoals
  - Activities and Events
  - Consequences and Impact
Types of Attacks

- Architectural / Design
- Implementation
- Operations
The flaws that lead to these kinds of attacks are the hardest to fix because they occur at such an early stage of the life cycle.

The design issue that can leave a system vulnerable are not necessarily mistakes.

- The design may achieve the desired functionality but not address potential security issues, e.g. telnet.
Attacks: Man-in-the-Middle or Eavesdropping

- Interception of a network transmission between two hosts.
- Masquerade as one of the two and then insert additional items into the exchange.

**Defense**
- Encryption
- Session checksums and shared secrets
Attacks: Race Condition

- Time step between parts of a process may be large enough to allow an attacker to compromise the system.

- The result often depends on the order in which parallel processes complete.
Attacks: Race Condition

Defense

- Use atomic (indivisible) operations as opposed to non-atomic (divisible) ones.

- Be aware of processes with security implications, e.g.
  - Opening a file
  - Invoking a subroutine
  - Checking a password
  - Verifying a username
Attacks: Replay

- The attacker captures an entire transaction between two parties and then replays part of it (impersonation).

**Defense**

- Encryption
- Introduce elements into the transmission that will differ from session to session such as a session number or date/time.
Attacks: Sniffer

A *sniffer* is a program that silently records all traffic sent over a LAN.

- It can record sensitive plaintext.

**Defense**
- Encryption
- Network level solutions
Sniffer Software

- Free packet sniffers are listed at http://www.netsecurity.about.com/library/blfreepacsniff.htm

- *Ethereal* is a free network sniffer.

- From their website: www.ethereal.com
  - *Ethereal* is a free network protocol analyzer for Unix and Windows. It allows you to examine data from a live network or from a capture file on disk. You can interactively browse the capture data, viewing summary and detail information for each packet.
Buffer Overflow

- This can occur in programs that allocate fixed length buffers (prevalent in the programming language C).

- *Example:* character string for user input.

- If the application does not perform adequate bounds checking then it may accept more characters than it can safely store.

- An attacker can cause the buffer to overflow on purpose and insert unauthorized executable commands into the code.
**Buffer Overflow**

- **Defense**
  - Do not use languages that allow this feature.
  - Avoid reading text strings of indeterminate length into fixed size buffers.
CERT Advisory CA-2003-12

There is a remotely exploitable vulnerability in sendmail that could allow an attacker to gain control of a vulnerable sendmail server. Due to a variable type conversion problem (char to signed int), sendmail may not adequately check the length of address tokens. A specially crafted email message could trigger a stack overflow.
Back Door

- A back door is code inserted by a programmer into a program to allow access control to be bypassed.

**Defense**
- Check all code for back doors - SQA.
A user may input malicious data that will be accepted by the program causing undesirable behaviour.

Occurs when a program does not check the input data for safety, e.g. web servers not checking for ../ in URLs.

**Defense**
- Use data checking tools.
Denial-of-Service

- This is caused by a high volume of service requests or input to a system.

- The system cannot handle this volume and consequently service is denied to legitimate users.
Denial-of-Service

Defense

- Make modest demands on system resources
- Make modest demands on system resources, such as disk space, number of open files, number of connections, etc.
- Monitor resource utilization and plan for excessive load.
Default Accounts

- Many systems are installed with easy to guess default user account names and passwords, e.g. guest/guest, field/service.

**Defense**

- Remove all default accounts after installation.
- Check for default after any installation of new software or upgrades.
Password Cracking

- There are cracking programs that use special algorithms and dictionaries of common words and phrases to generate guesses.

- These guesses are checked against the password file or against login accounts.

**Defense**
- Require robust passwords.
Factors that work against Secure Code

- Technical
- Psychological
- Real-world
Technical Factors

- **Mistake by side-effect**
  - Happens when the designer/programmer does not fully understand what a particular function does, e.g. allocating memory, reading/writing to disk.
  - Happens when components are combined and errors or misunderstands are exposed.

- **Complexity!!!!!!**
Real-World

- Who is the source of the source code?
- Lack of software engineering education.
  - Standards and metrics are needed.
- Production pressures.
  - Testing time is limited.
- Security is not a priority.
  - Little impact on bottom line.
  - Less security = more features.
Psychological

- Tragedy of the Commons

- Individual short-term gain = Group long-term loss.


- [http://www.garretthardinsociety.org/articles/art_tragedy_of_the_commons.html](http://www.garretthardinsociety.org/articles/art_tragedy_of_the_commons.html)
A Case Study: rlogin

The Story of rlogin

- rlogin establishes a remote login session from its user’s terminal to a remote host computer.

- It passes the terminal type description from the local host to the remote host.

- rlogin passes the current terminal definition as identified by the TERM environment variable.
The Coding Defect in rlogin

- The value of the TERM environment variable is copied without due care to an internal buffer.
  - Buffer overflow !!!!
  - The buffer is a variable local to the main subroutine; the local host’s subroutine linkage information can be overwritten with data from the TERM environment variable.
  - This can transfer control to an arbitrary memory address
More Bad News

- rlogin requires set-user-id root privileges so that it can obtain a port in the range 0-1023.

- When the “smashed stack” instructions are executed they run in full root mode since the setuid call to leave root mode is later in the code.
The Bad Code

```c
char term[1024];
(void) strcpy (term, (p = getenv("TERM"))? : "network");
Rem = rcmd ( &host, sp−>s_port, pw−>pw_name, user, term, 0 );
(void) setuid ( uid );
```
Exploitation

- A well crafted TERM environment variable is usually constructed to smash the stack and start a copy of /bin/sh.

- The attacker can now run any command as root.
Mitigation Strategies

- Non-executable Stack Regions
  - Remove execution permission from the stack segment of every process on a UNIX system.
  - But, some legitimate uses of an executable stack exist.
  - Not all processor MMUs can enforce this.
  - A corrupted stack could still transfer control to a non-stack-based address.
Mitigation Strategies

- **Truncating the Data**
  
  - Some data may be lost and the reaction of the remote host to this is unpredictable.
  - Replace `strcpy` with `strncpy` and insert the NULL terminator at the end of the term string.
  - Replace `rlogin` with a wrapper program that truncates the `TERM` variable before passing it to `rlogin` (*Does not rely on changing the source code*).
Mitigation Strategies

- Inspect Data for Length and Content
  - Inspect both the TERM variable’s length and its content.
  - Data that does not fit or is incorrect should be replaced by a meaningful default value to ensure predictable results.
  - This can be implemented via a wrapper or via changes to the source code.
Part II

Approaches to Secure Computing
Defensive programming practices strive to allow the programmer to implement the necessary functionality without injecting coding defects that will have to be repaired later in the life cycle.

Assume that bad input to a program is not just an error but can be malicious in intent.

- Input includes data in files, arguments, environment variables, credentials, open file descriptors, data streams, and system resources.
Practice 1: Trusting Untrustworthy Data

- When data crosses a boundary it must be considered untrustworthy and be made trustworthy before being used.

- A boundary is an imaginary line separating two potentially competing domains.
- One of the boundaries in the case of rlogin was between the user and the program.
- This is the case when a user inputs data or passes arguments to a program.
The Practice

To make untrustworthy data trustworthy:

- Identify the boundaries in a program.
- Identify data that crosses boundaries.
- Examine the data for correct form and substitute meaningful default values for nonconforming data.
Practice 2: Shedding Privileges

- rlogin has to run as root to gain access to a reserved port as part of the authentication process.

- Actually this is a weak authentication scheme and its use of root privilege makes rlogin’s security flaw extremely dangerous.
The Practice

- Use privileges at the beginning of your program.
- Give up privileges immediately after you are done with them.
- Give up privileges in a manner so that they cannot be reclaimed.
  - Temporarily giving up privileges only slows down the attacker.
The Practice

- Write concise privileged code so that it can be thoroughly inspected for defects.

- Isolate code segments that require privileges.
  - Change the programming interfaces to vendor-provided software.
Other Sources of Information on Secure Code

- Fyodor’s Good Reading List on insecure.org (http://www.insecure.org/reading.html)

- Must reads are
  - Mudge’s tutorial on writing buffer overflows
  - Smashing the Stack for Fun and Profit
  - Murphy’s Law and Computer Security
  - Insertion, Evasion and Denial of Service: Eluding Network Invasion Detection
Top 10 Secure Coding Practices

Top 10 Secure Coding Practices

1. **Validate input.**
   - Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files [Seacord 05].

2. **Heed compiler warnings.**
   - Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.
3. **Architect and design for security policies.**

- Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
4. **Keep it simple.**

   Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
Top 10 Secure Coding Practices

5. **Default deny.**
   - Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted [Saltzer 74, Saltzer 75].

6. **Adhere to the principle of least privilege.**
   - Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75].
7. **Sanitize data sent to other systems.**

Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.
8. **Practice defense in depth.**

Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05].
9. **Use effective quality assurance techniques.**

- Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Penetration testing, fuzz testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05].

10. **Adopt a secure coding standard.**

- Develop and/or apply a secure coding standard for your target development language and platform.
1. **Define security requirements.**

   Identify and document security requirements early in the development life cycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements. When security requirements are not defined, the security of the resulting system cannot be effectively evaluated.
2. **Model threats.**

   - Use threat modeling to anticipate the threats to which the software will be subjected. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases [Swiderski 04].
(Saltzer 74) Saltzer, J. H. "Protection and the Control of Information Sharing in Multics." Communications of the ACM 17, 7 (July 1974): 388-402.

