Designing and Writing Secure Application Code

John Mitchell

Outline

- General principles
  - Least privilege, defense in depth, ...
- Example
  - Sendmail vs qmail
- Tools for secure coding
  - Type-safe programming languages
  - Purify
  - Perl tainting
  - Code analysis algorithms
  - Run-time monitoring (another lecture)

General topics in this course

- Vulnerabilities
  - How hackers break into systems
    - Circumvent security mechanisms (e.g., dictionary attack)
    - Use code for purpose it was not intended (buffer overflow)
- Defensive programming
  - Build all software with security in mind
  - Make sure video game is not a boot loader
- Security Mechanisms
  - Authentication
  - Access control
  - Network protocols

Before you start building ...

- What are the security requirements?
  - Confidentiality (secrets remain secret)
  - Integrity (meaning preserved)
  - Availability
  - Accountability
- What threats are possible?
- Who do you trust / not trust?

Security = preserve properties against attack

General advice

- Compartmentalization
  - Principle of least privilege
  - Minimize trust relationships
- Defense in depth
  - Use more than one security mechanism
  - Secure the weakest link
  - Fail securely
- Keep it simple
- Consult experts
  - Don't build what you can easily borrow/steal
  - Open review is effective and informative

Compartmentalization

- Divide system into modules
  - Each module serves a specific purpose
  - Assign different access rights to different modules
    - Read/write access to files
    - Read user or network input
    - Execute privileged instructions (e.g., Unix root)
  - Principle of least privilege
    - Give each module only the rights it needs
Compartmentalization (II)

Example
- Sendmail runs as root
  - Root privilege needed to bind port 25
  - No longer needed after port bind established
- But most systems keep running as root
  - Root privileges needed later to write to user mailboxes
- Will look at qmail for better security design

Minimize trust relationships
- Clients, servers should not trust each other
  - Both can get hacked
- Trusted code should not call untrusted code

Example: .NET

class NativeMethods
{
  // This is a call to unmanaged code. Executing this method requires
  // the UnmanagedCode security permission. Without this permission,
  // an attempt to call this method will throw a SecurityException:
  [DllImport("msvcrt.dll")]
  public static extern int puts(string str);
  [DllImport("msvcrt.dll")]
  internal static extern int _flushall();
}

Code denies permission not needed

{[SecurityPermission(SecurityAction.Deny, Flags =
  SecurityPermissionFlag.UnmanagedCode)]}
  private static void MethodToDoSomething()
  {
    try
    {
      Console.WriteLine("... ");
      SomeOtherClass.method();
    }
    catch (SecurityException)
    {
      ...
    }
  }

Defense in Depth

- Failure is unavoidable – plan for it
- Have a series of defenses
  - If an error or attack is not caught by one
    mechanism, it should be caught by another
- Examples
  - Firewall + network intrusion detection
  - SSH + Tripwire
- Fail securely
  - Many, many vulnerabilities are related to error
    handling, debugging or testing features, error
    messages

Secure the weakest link

- Think about possible attacks
  - How would someone try to attack this?
  - What would they want to accomplish?
- Find weakest link(s)
  - Crypto library is probably pretty good
  - Is there a way to work around crypto?
    - Data stored in encrypted form; where is key stored?
- Main point
  - Do security analysis of the whole system
  - Spend your time where it matters

Keep It Simple

- Use standard, tested components
  - Don’t implement your own cryptography
- Don’t add unnecessary features
  - Extra functionality ⇒ more ways to attack
- Use simple algorithms that are easy to verify
  - A trick that may save a few instructions may
    ⇒ Make it harder to get the code right
  - Make it harder to modify and maintain code
Promote Privacy

- Discard information when no longer needed
  - No one can attack system to get information
- Examples
  - Don't keep log of old session keys
  - Delete firewall logs
  - Don't run unnecessary services (fingerd)
- Hiding sensitive information is hard
  - Information in compiled binaries can be found
  - Insider attacks are common
  - Security by obscurity doesn't work!!!

Don't reinvent the wheel

- Consult experts
- Allow public review
- Use software, designs that other have used
- Examples
  - Bad use of crypto: 802.11b
  - Protocols without expert review: 802.11i
  - Use standard url parser, crypto library, good random number generator, ...

Example: Mail Transport Agents

- Sendmail
  - Complicated system
  - Source of many vulnerabilities
- Qmail
  - Simpler system designed with security in mind
  - Gaining popularity

Qmail was written by Dan Bernstein, starting 1995
$500 reward for successful attack; no one has collected

Market share

<table>
<thead>
<tr>
<th>Year</th>
<th>Sendmail</th>
<th>Qmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>1997</td>
<td>76%</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>63%</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>55%</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>47%</td>
<td>9%</td>
</tr>
<tr>
<td>2002</td>
<td>42%</td>
<td>17%</td>
</tr>
</tbody>
</table>

Simplified Mail Transactions

- Message composed using an MUA
- MUA gives message to MTA for delivery
  - If local, the MTA gives it to the local MDA
  - If remote, transfer to another MTA

Stanford Sendmail Vulnerability

Sent: Tuesday, March 04, 2003 1:12 PM
To: unix-info@lists.Stanford.EDU
Subject: Stanford ITSS Security Alert: sendmail Header Processing Vulnerability

sendmail is the most popular Mail Transfer Agent (MTA) program in use on the Internet, ...

sendmail contains an error in one of the security checks it employs on addresses in its headers, which may allow an attacker to execute malicious code within the sendmail security context, usually root...

All users of sendmail should patch immediately...
**Example: qmail**

- **Least privilege**
  - Each module uses least privileges necessary
  - Only one setuid program
    - setuid to one of the other qmail user IDs, not root
    - No setuid root binaries
  - Only one run as root
    - Spawns the local delivery program under the UID and GID of the user being delivered to
    - No delivery to root
    - Always changes effective uid to recipient before running user-specified program

- **Other secure coding ideas**

**Structure of qmail**

- Splits mail msg into 3 files
  - Message contents
  - 2 copies of header, etc.
  - Signals qmail-send

- qmail-send signals
  - qmail-lspawn if local
  - qmail-remote if remote

- qmail-lspawn
  - Spawns qmail-local
  - qmail-local runs with ID of user receiving local mail

- qmail-local
  - Handles alias expansion
  - Delivers local mail
  - Calls qmail-queue if needed

- Other incoming mail

- Incoming SMTP mail
Structure of qmail

- qmail-smtpd
- qmail-local
- qmail-remote
- qmail-rspawn
- qmail-send
- qmail-inject
- qmail-queue

Least privilege

- qmail-smtpd
- qmail-local
- qmail-remote
- qmail-lspawn
- qmail-rspawn
- qmail-send
- qmail-inject
- qmail-queue

Principles, sendmail vs qmail

- Do as little as possible in setuid programs
  - Of 20 recent sendmail security holes, 11 worked only because the entire sendmail system is setuid
  - Only qmail-queue is setuid
    - Its only function is add a new message to the queue
- Do as little as possible as root
  - The entire sendmail system runs as root
    - Operating system protection has no effect
  - Only qmail-start and qmail-lspawn run as root.

Principles, sendmail vs qmail

- Programs and files are not addresses
  - sendmail treats programs and files as addresses
    - "sendmail goes through horrendous contortions trying to keep track of whether a local user was responsible for an address. This has proven to be an unmitigated disaster" (DJB)
  - qmail programs and files are not addresses
    - "The local delivery agent, qmail-local, can run programs or write to files as directed by ~/.qmail, but it's always running as that user. Security impact: qmail, like shcnc and exec and various other files, means that anyone who can write arbitrary files as a user can execute arbitrary programs as that user. That's it." (DJB)

Keep it simple

- Parsing
  - Limited parsing of strings
    - Minimizes risk of security holes from configuration errors
- Libraries
  - Avoid standard C library, stdio
    - "Write bug-free code" (DJB)
- Don't repeat functionality
  - One simple mechanism handles forwarding, aliasing, and mailing lists (instead of 3)
  - Single delivery mode instead of a selection
Comparison

<table>
<thead>
<tr>
<th>Package</th>
<th>Lines</th>
<th>Words</th>
<th>Chars</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>qmail-1.01</td>
<td>16028</td>
<td>44331</td>
<td>370123</td>
<td>288</td>
</tr>
<tr>
<td>sendmail-8.8.8</td>
<td>52830</td>
<td>179608</td>
<td>1218116</td>
<td>53</td>
</tr>
<tr>
<td>zmailer-2.2e10</td>
<td>57595</td>
<td>205524</td>
<td>1423624</td>
<td>227</td>
</tr>
<tr>
<td>small-3.2</td>
<td>62331</td>
<td>246140</td>
<td>1701112</td>
<td>151</td>
</tr>
<tr>
<td>exim-1.90</td>
<td>67778</td>
<td>272084</td>
<td>2092351</td>
<td>127</td>
</tr>
</tbody>
</table>

Additional general advice  [Wheeler]

- Validate input
- Structure your program security
  - Secure the interface, minimize privileges
  - Make the initial configuration and defaults safe
  - Avoid race conditions
  - Trust only trustworthy channels
- Carefully call out to other resources
  - Check all system calls and return values

Program Component

- Avoid buffer overflow
- Structure your program security
- Validate input
- Call out to other code carefully
- Respond judiciously

See Wheeler’s book on web

Summary from [Wheeler]

- Validate all your inputs
  - Command line inputs, environment variables, CGI inputs, ...
  - Don’t just reject “bad” input, define “good” and reject all else
- Avoid buffer overflow
- Structure your program security
  - Secure the interface, minimize privileges
  - Make the initial configuration and defaults safe
  - Avoid race conditions
  - Trust only trustworthy channels
- Carefully call out to other resources
  - Check all system calls and return values

Additional reading

- Contents
  - Contemporary security
    - Need for secure systems, proactive security process
  - Secure Coding Techniques
    - The buffer overrun
    - Determining appropriate access control
    - Running with least privilege
    - Protecting secret data
    - All input is evil
  - More Secure Coding techniques
  - Special Topics
    - Testing, code review, secure installation, privacy

Tools for producing secure code

- C vs type safe languages
  - Buffer overflows are array bounds violations
  - Java, ML, ... check array bounds, prevent overflow
- Purify
  - Find memory errors on the heap
- Perl tainting
  - Track use of untrusted input
- Automated code analysis tools
  - Can catch many kinds of errors

Purify

- Goal
  - Instrument a program to detect run-time memory errors (out-of-bounds, use-before-init) and memory leaks
- Technique
  - Works on relocatable object code
    - Link to modified malloc that provides tracking tables
  - Memory access errors: insert instruction sequence before each load and store instruction
  - Memory leaks: GC algorithm
Perl tainting

- Run-time checking of Perl code
  - Perl used for CGI scripts, security sensitive
  - Taint checking stops some potentially unsafe calls
- Tainted strings
  - User input: Values derived from user input
  - Except result of matching against untainted string
- Prohibited calls
  - print $form_data{"email"} . "\n";
    - OK since print is safe (???)
  - system("mail " . $form_data{"email"});
    - Flagged system call with user input as argument

Safe Perl mail command (?)

- Check email string against pattern and parse
  $email = $form_data{"email"};
  if ( $email =~ /(\w{1}\[\w-\.]*)\@[\w-\.]+/) {
      $email = "$1"@$2";
  } else { warn ("TAINTED DATA SENT BY ..."Atual error in Linux: raid5 driver disables interrupts, and then if it fails to allocate buffer. returns with them disabled. This kernel deadlock is actually hidden by an immediate segmentation fault since the callers dereference the pointer without checking for NULL.

Automated code analysis for C

- Example tool
  - Used modified compiler to find over 100 security holes in Linux and BSD
  - http://www.stanford.edu/~engler/

- Benefit
  - Capture recommended practices, known to experts, in tool available to all

Checking secure software

- Many rules for writing secure code
  - "sanitize user input before using it"
  - "check permissions before doing operation X"
- How to find errors?
  - Formal verification: rigorous
    - costly, expensive. *Very* rare to do for software
  - Testing:
    - simple, few false positives
      - requires running code: doesn't scale & can be impractical
  - Manual inspection
    - flexible
      - erratic & doesn't scale well.
  - What to do??

Sanitize integers before use

Warn when unchecked integers from untrusted sources reach trusting sinks

- Warn when unchecked integers from untrusted sources reach trusting sinks
- Uses checker
- Actual error in Linux: raid5 driver disables interrupts, and then if it fails to allocate buffer, returns with them disabled. This kernel deadlock is actually hidden by an immediate segmentation fault since the callers dereference the pointer without checking for NULL.

Metacomilation (MC)

- Analyze compiler data structure to check code
  - Extensions dynamically linked into GNU gcc compiler
  - Applied down all paths in input program source
  - E.g., extension to check user input

Linux 2.4.5: drivers/usb/see401.c

Actual error in Linux: raid5 driver disables interrupts, and then if it fails to allocate buffer, returns with them disabled. This kernel deadlock is actually hidden by an immediate segmentation fault since the callers dereference the pointer without checking for NULL.

Linux: 125 errors, 24 false; BSD: 12 errors, 4 false
Example security holes

Remote exploit, no checks

```c
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
    msg = skb->data;
    ...
    memcpy(cmd.parm.setup.phone, msg->msg.connect_ind.addr.num, msg->msg.connect_ind.addr.len - 1);
```

User-pointer inference

Problem: which are the user pointers?
- Hard to determine by dataflow analysis
- Easy to tell if kernel believes pointer is from user!

Belief inference
- "*p" implies safe kernel pointer
- "copyin(p)/copyout(p)" implies dangerous user ptr
- Error: pointer p has both beliefs.

Implementation: 2 pass checker
- Inter-procedural: compute all tainted pointers
- Local pass to check that they are not dereferenced

Example security holes

Missed lower-bound check:

```c
/* 2.4.5/drivers/char/drm/i810_dma.c */
if(copy_from_user(&d, arg, sizeof(arg)))
    return -EFAULT;
if(d.idx > dma->buf_count)
    return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
```

Results for BSD and Linux

All bugs released to implementers; most serious fixed

<table>
<thead>
<tr>
<th>Violation</th>
<th>Linux Bug Fixed</th>
<th>BSD Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial of service</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Read arbitrary memory</td>
<td>45</td>
<td>17</td>
</tr>
<tr>
<td>Corrupt memory</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>Denial of control of system</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>125</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Violation</th>
<th>Linux Bug Fixed</th>
<th>BSD Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local bugs</td>
<td>109</td>
<td>12</td>
</tr>
<tr>
<td>Global bugs</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Bugs from inferred ints</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>False positives</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Number of checks</td>
<td>~3500</td>
<td>594</td>
</tr>
</tbody>
</table>

Conclusions

- Security takes extra effort
  - Know your security goals
  - Design with security in mind
    - Compartmentalize, least privilege
    - Minimize setuid, root
  - Implement carefully
    - Keep it simple
    - Think about attacks; secure the weakest link
  - Use tools that detect common coding problems
    - There are also tools that can analyze designs, but that's another story (harder to use, current research)