Secure Programming, Continued
Computer Security
Peter Reiher
December 4, 2014

Outline

- Introduction
- Principles for secure software
- Major problem areas

Example Problem Areas

- Buffer overflows
- Error handling
- Access control issues
- Race conditions
- Use of randomness
- Proper use of cryptography
- Trust
- Input verification
- Variable synchronization
- Variable initialization

Error Handling

- Error handling code often gives attackers great possibilities
- It’s rarely executed and often untested
- So it might have undetected errors
- Attackers often try to compromise systems by forcing errors

A Typical Error Handling Problem

- Not cleaning everything up
- On error conditions, some variables don't get reset
- If error not totally fatal, program continues with old values
- Could cause security mistakes
  - E.g., not releasing privileges when you should

Some Examples

- Remote denial of service attack on Apache HTTP server due to bad error handling (2010)
- Internet Explorer arbitrary code execution flaw (2007)
  - Use-after-free bug in script error handling code

Checking Return Codes

- A generalization of error handling
- Always check return codes
- A security program manager for Microsoft said this is his biggest problem
- Very dangerous to bull ahead if it turns out your call didn't work properly
- Example: Nagios XI didn't check the return value of `setuid()` call, allowing privilege escalation

Access Control Issues

- Programs usually run under their user's identity with his privileges
- Some programs get expanded privileges
  - Setuid programs in Unix, e.g.
- Poor programming here can give too much access

An Example Problem

- A program that runs setuid and allows a shell to be forked
  - Giving the caller a root environment in which to run arbitrary commands
- Buffer overflows in privileged programs usually give privileged access

A Real World Example

- `/sbin/dump` from NetBSD
- Ran setgid as group `tty`
  - To notify sysadmins of important events
  - Never dropped this privilege
- Result: `dump` would start program of user's choice as user `tty`
  - Allowing them to interact with other user's terminals

What To Do About This?

- Avoid running programs setuid
  - Or in other OSs' high privilege modes
- If you must, don't make them root-owned
  - Remember, least privilege
- Change back to the real caller as soon as you can
  - Limiting exposure
- Use virtualization to compartmentalize

Virtualization Approaches

- Run stuff in a virtual machine
  - Only giving access to safe stuff
- Hard to specify what's safe
- Hard to allow safe interactions between different VMs
- VM might not have perfect isolation

Race Conditions

- A common cause of security bugs
- Usually involve multiprogramming or multithreaded programs
- Caused by different threads of control operating in unpredictable fashion
  - When programmer thought they'd work in a particular order

What Is a Race Condition?

- A situation in which two (or more) threads of control are cooperating or sharing something
- If their events happen in one order, one thing happens
- If their events happen in another order, something else happens
- Often the results are unforeseen

Security Implications of Race Conditions

- Usually you checked privileges at one point
- You thought the next lines of code would run next
  - So privileges still apply
- But multiprogramming allows things to happen in between

The TOCTOU Issue

- Time of Check to Time of Use
- Have security conditions changed between when you checked?
- And when you used it?
- Multiprogramming issues can make that happen
- Sometimes under attacker control

A Short Detour

- In Unix, processes can have two associated user IDs
  - Effective ID
  - Real ID
- Real ID is the ID of the user who actually ran it
- Effective ID is current ID for access control purposes
- Setuid programs run this way
- System calls allow you to manipulate it

Effective UID and Access Permissions

- Unix checks accesses against effective UID, not real UID
- So setuid program uses permissions for the program's owner
  - Unless relinquished
- Remember, root has universal access privileges

An Example

- Code from Unix involving a temporary file
- Runs setuid root

```
res = access("/tmp/userfile", R_OK);
If (res != 0)
  die("access");
fd = open("/tmp/userfile",O_RDONLY);
```

What's (Supposed to Be) Going on Here?

- Checked access on `/tmp/userfile` to make sure user was allowed to read it
  - User can use links to control what this file is
- `access()` checks real user ID, not effective one
  - So checks access permissions not as root, but as actual user
- So if user can read it, open file for read
  - Which root is definitely allowed to do
- Otherwise exit

What's Really Going On Here?

- This program might not run uninterrupted
- OS might schedule something else in the middle
- In particular, between those two lines of code

How the Attack Works

- Attacker puts innocuous file in
  `/tmp/userfile`
- Calls the program
- Quickly deletes file and replaces it with link to sensitive file
  - One only readable by root
- If timing works, he gets secret contents

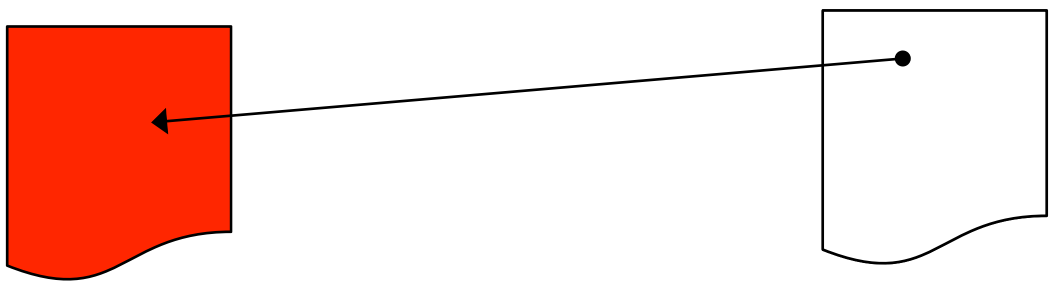
The Dynamics of the Attack

Success!

Let's try that again!

One more time!

/etc/secretfile /tmp/userfile



1. Run program
2. Change file

```
res = access("/tmp/userfile", R_OK);
if (res != 0)
      die("access");
fd = open("/tmp/userfile",O_RDONLY);
```

How Likely Was That?

- Not very
  - The timing had to be just right
- But the attacker can try it many times
  - And may be able to influence system to make it more likely
- And he only needs to get it right once
- Timing attacks of this kind can work
  - Google Chrome had one in 2011
- The longer between check and use, the more dangerous

Some Types of Race Conditions

- File races
  - Which file you access gets changed
- Permissions races
  - File permissions are changed
- Ownership races
  - Who owns a file changes
- Directory races
  - Directory hierarchy structure changes

Preventing Race Conditions

- Minimize time between security checks and when action is taken
- Be especially careful with files that users can change
- Use locking and features that prevent interruption, when possible
- Avoid designs that require actions where races can occur

Randomness and Determinism

- Many pieces of code require some randomness in behavior
- Where do they get it?
- As earlier key generation discussion showed, it's not that easy to get

Pseudorandom Number Generators

- PRNG
- Mathematical methods designed to produce strings of random-like numbers
- Actually deterministic
  - But share many properties with true random streams of numbers

Attacks on PRNGs

- Cryptographic attacks
  - Observe stream of numbers and try to deduce the function
- State attacks
  - Attackers gain knowledge of or influence the internal state of the PRNG

An Example

- ASF Software's Texas Hold'Em Poker
- Flaw in PRNG allowed cheater to determine everyone's cards
  - Flaw in card shuffling algorithm
  - Seeded with a clock value that can be easily obtained

Another Example

- Flaw in Android random number generator in 2013
- Left Bitcoin wallets in that platform vulnerable to theft
  – By making it much easier to deduce a secret key that used the RNG

How to Do Better?

- Use hardware randomness, where available
- Use high quality PRNGs
  - Preferably based on entropy collection methods
- Don’t use seed values obtainable outside the program

Proper Use of Cryptography

- Never write your own crypto functions if you have any choice
  - Another favorite piece of advice from industry
- Never, ever, design your own encryption algorithm
  - Unless that's your area of expertise
- Generally, rely on tried and true stuff
  - Both algorithms and implementations

Proper Use of Crypto

- Even with good crypto algorithms (and code), problems are possible
- Proper use of crypto is quite subtle
- Bugs possible in:
  - Choice of keys
  - Key management
  - Application of cryptographic ops

An Example

- An application where RSA was used to distribute a triple-DES key
- Seemed to work fine
- Someone noticed that part of the RSA key exchange was always the same
  - That's odd . . .

What Was Happening?

- Bad parameters were handed to the RSA encryption code
- It failed and returned an error
- Which wasn't checked for
  - Since it "couldn't fail"
- As a result, RSA encryption wasn't applied at all
- The session key was sent in plaintext . . .

Trust Management

- Don't trust anything you don't need to
- Don't trust other programs
- Don't trust other components of your program
- Don't trust users
- Don't trust the data users provide you

Trust

- Some trust required to get most jobs done
- But determine how much you must trust the other
  - Don't trust things you can independently verify
- Limit the scope of your trust
  - Compartmentalization helps
- Be careful who you trust

Two Important Lessons

1. Many security problems arise because of unverified assumptions
   – You think someone is going to do something he actually isn’t
2. Trusting someone doesn’t just mean trusting their honesty
   – It means trusting their caution, too

Input Verification

- Never assume users followed any rules in providing you input
- They can provide you with anything
- Unless you check it, assume they've given you garbage
  - Or worse
- Just because the last input was good doesn't mean the next one will be

Treat Input as Hostile

- If it comes from outside your control and reasonable area of trust
- Probably even if it doesn't
- There may be code paths you haven't considered
- New code paths might be added
- Input might come from new sources

For Example

- Shopping cart exploits
- Web shopping carts sometimes handled as a cookie delivered to the user
- Some of these weren’t encrypted
- So users could alter them
- The shopping cart cookie included the price of the goods . . .

What Was the Problem?

- The system trusted the shopping cart cookie when it was returned
  - When there was no reason to trust it
- Either encrypt the cookie
  - Making the input more trusted
  - Can you see any problem with this approach?
- Or scan the input before taking action on it
  - To find refrigerators being sold for 3 cents

Variable Synchronization

- Often, two or more program variables have related values
- Common example is a pointer to a buffer and a length variable
- Are the two variables always synchronized?
- If not, bad input can cause trouble

An Example

- From Apache web server
- `cdata` is a pointer to a buffer
- `len` is an integer containing the length of that buffer
- Programmer wanted to get rid of leading and trailing white spaces

The Problematic Code

```
while (apr_isspace(*cdata))
  ++cdata;
while (len-- >0 &&
  apr_isspace(cdata[len]))
  continue;
cdata[len+1] = '/0';
```

- `len` is not decremented when leading white spaces are removed
- So trailing white space removal can overwrite end of buffer with nulls
- May or may not be serious security problem, depending on what's stored in overwritten area

Variable Synchronization and Heartbleed

- Heartbleed was essentially a variable synchronization bug
- One variable was an array of characters
- A second variable was supposedly the length of the array
- Bug depended on not verifying that the provided length matched the array size

Variable Initialization

- Some languages let you declare variables without specifying their initial values
- And let you use them without initializing them
  - E.g., C and C++
- Why is that a problem?

A Little Example

```
main()
{

foo();
bar();
}

foo()
{
        int a;
        int b;
        int c;

        a = 11;
        b = 12;
        c = 13;
}
```

```
bar()
{
        int aa;
        int bb;
        int cc;

        printf("aa = %d\n",aa);
        printf("bb = %d\n",bb);
        printf("cc = %d\n",cc);
}
```

What's the Output?

```
lever.cs.ucla.edu[9]./a.out
aa = 11
bb = 12
cc = 13
```

- Perhaps not exactly what you might want

Why Is This Dangerous?

- Values from one function “leak” into another function
- If attacker can influence the values in the first function,
- Maybe he can alter the behavior of the second one

Variable Cleanup

- Often, programs reuse a buffer or other memory area
- If old data lives in this area, might not be properly cleaned up
- And then can be treated as something other than what it really was
- E.g., bug in Microsoft TCP/IP stack
  - Old packet data treated as a function pointer
- Part of the Heartbleed problem, too
  - Buffer not currently in use (but addressable) contained old keys and passwords

Some Other Problem Areas

- Handling of data structures
  - Indexing errors in DAEMON Tools, Oracle JRE
- Arithmetic issues
  - Integer overflow in Microsoft Office graphics library
  - Signedness error in XnView
- Errors in flow control
  - Samba error that causes loop to use wrong structure
- Off-by-one errors
  - Denial of service flaw in Clam AV

Yet More Problem Areas

- Memory management errors
  – Use-after-free error in Internet Explorer
- Null pointer dereferencing
  – Xarrow SCADA system denial of service
- Side effects
- Punctuation errors
- There are many others

Why Should You Care?

- A lot of this stuff is kind of exotic
- Might seem unlikely it can be exploited
- Sounds like it would be hard to exploit without source code access
- Many examples of these bugs probably unexploitable

So . . .?

- Well, that's what everyone thinks before they get screwed
- "Nobody will find this bug"
- "It's too hard to figure out how to exploit this bug"
- "It will get taken care of by someone else"
  - Code auditors
  - Testers
  - Firewalls

That's What They Always Say

- Before their system gets screwed
- Attackers can be very clever
  - Maybe more clever than you
- Attackers can work very hard
  - Maybe harder than you would
- Attackers may not have the goals you predict

But How to Balance Things?

- You only have a certain amount of time to design and build code
- Won’t secure coding cut into that time?
- Maybe
- But less if you develop code coding practices
- If you avoid problematic things, you’ll tend to code more securely

Some Good Coding Practices

- Validate input
- Be careful with failure conditions and return codes
- Avoid dangerous constructs
  - Like C input functions that don't specify amount of data
- Keep it simple