Secure Programming
Computer Security
Peter Reiher
February 18, 2021

Outline

- Introduction
- Principles for secure software
- Choosing technologies
- Major problem areas
- Evaluating program security

Introduction

- How do you write secure software?
- Basically, define security goals
- And use techniques that are likely to achieve them
- Ideally, part of the whole process of software development
  - Not just some tricks programmers use

Designing for Security

- Often developers design for functionality
  - "We'll add security later"
- Security retrofits have a terrible reputation
  - Insecure designs offer too many attack opportunities
- Designing security from the beginning works better

Defining Security Goals

- Think about which security properties are relevant to your software
  - Does it need limited access?
  - Privacy issues?
  - Is availability important?
- And the way it interacts with your environment
  - Even if it doesn't care about security, what about the system it runs on?

Security and Other Goals

- Security is never the only goal piece of software
- Usually not the primary goal
- Generally, secure software tha meet its other goals is a failure
- Consider the degree of security required as an issue of *risk*

Many engineering disciplines must deal with risk.

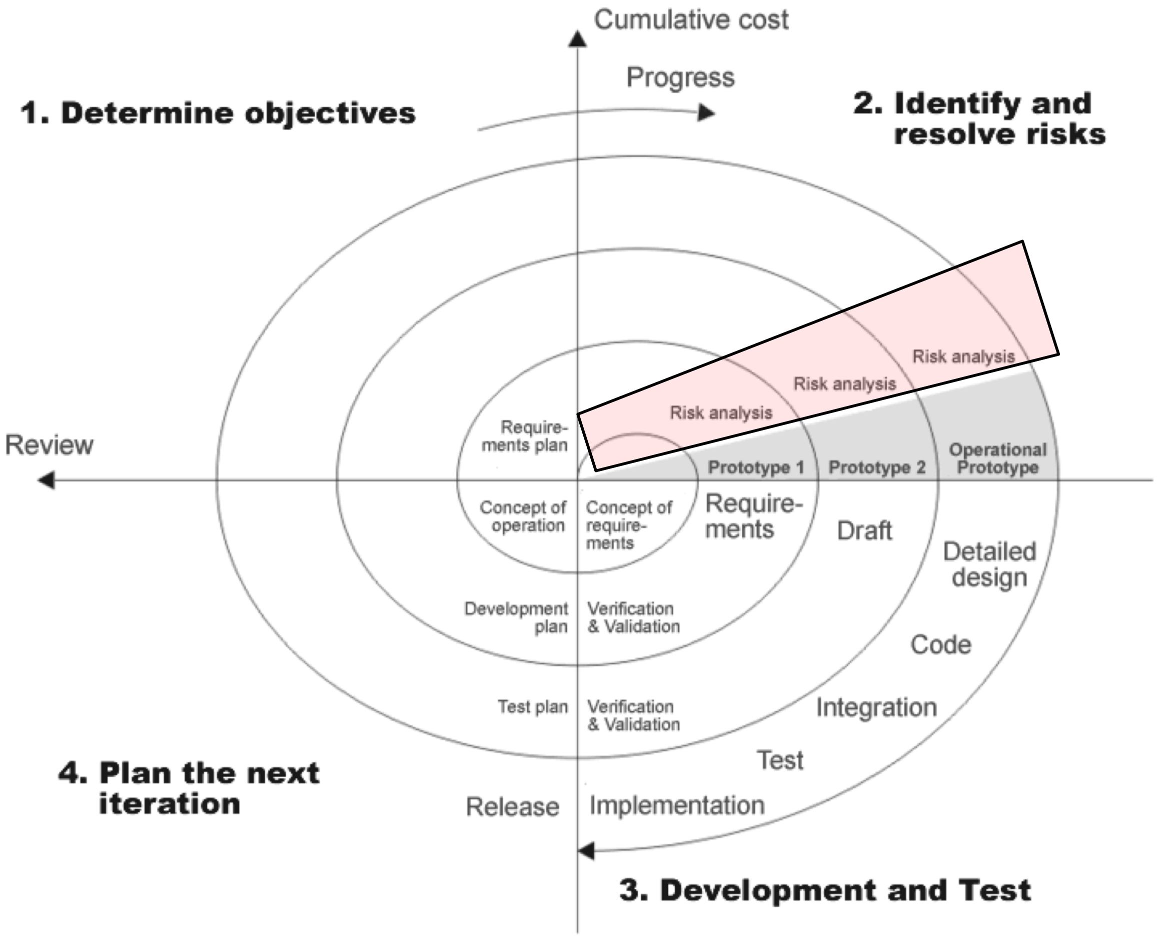
Managing Software Security Risk

- How much risk can this software tolerate?
- What compromises can you make to minimize that risk?
  - Often other goals conflict with security
  - E.g., should my program be more usable or require strong authentication?
- Considering tradeoffs in terms of risks can clarify what you need to do

Risk Management and Software Development

- Should consider security risk as part of your software development model
- E.g., in spiral model, add security risk analysis phase to the area of spiral where you evaluate alternatives
- Considering security and risks early can avoid pitfalls later
- Returning to risk when refining is necessary

Incorporating Security Into Spiral Model of SW Development



Include security in the risks you consider

At all passes through the spiral

But How Do I Determine Risk?

- When you're just thinking about a big new program, how can you know about its risks?
- Well, do the best you can
  - Apply your knowledge and experience
  - Really think about the issues and problems
  - Use a few principles and tools we'll discuss
- That puts you ahead of 95% of all developers
- You can't possibly get it all right, but any attention to risk is better than none

Design and Security Experts

- Someone on a software development team should understand security
  - The more they understand it, the better
  - Ideally, someone on team should have explicit security responsibility
- Experts should be involved in all phases
  - Starting from design

Principles for Secure Software

- Following these doesn’t guarantee security
- But they touch on the most commonly seen security problems
- Thinking about them is likely to lead to more secure code

1. Secure the Weakest Link

- Don’t consider only a single possible attack
- Look at all possible attacks you can think of
- Concentrate most attention on most vulnerable elements

For Example,

- Those attacking your web site are not likely to break transmission cryptography
  - Switching from RSA to elliptic probably doesn't address your weakest link
- Attackers are more likely to use a buffer overflow to break in
  - And read data before it's encrypted
  - Prioritize preventing that

2. Practice Defense in Depth

- Try to avoid designing software so failure anywhere compromises everything
- Also try to protect data and applications from failures elsewhere in the system
- Don’t let one security breach give away everything

For Example,

- You write a routine that validates all input properly
- All other routines that are supposed to get input should use that routine
- Worthwhile to have those routines also do some validation
  - What if there's a bug in your general routine?
  - What if someone changes your code so it doesn't use that routine for input?

3. Fail Securely

- Security problems frequently arise when programs fail
- They often fail into modes that aren't secure
- So attackers cause them to fail
  - To see if that helps them
- So make sure that when ordinary measures fail, the fallback is secure

For Example,

- A major security flaw in typical Java RMI implementations
- If server wants to use security protocol client doesn't have, what happens?
  - Client downloads it from the server
  - Which it doesn't trust yet . . .
- Malicious entity can force installation of compromised protocol

4. Use Principle of Least Privilege

- Give minimum access necessary
- For the minimum amount of time required
- Always possible that the privileges you give will be abused
  - Either directly or through finding a security flaw
- The less you give, the lower the risk

For Example,

- Say your web server interacts with a backend database
- It only needs to get certain information from the database
  - And uses access control to determine which remote users can get it
- Set access permissions for database so server can <u>only</u> get <u>that</u> data
- If web server hacked, only part of database is at risk

5. Compartmentalize

- Divide programs into pieces
- Ensure that compromise of one piece does not automatically compromise others
- Set up limited interfaces between pieces
  - Allowing only necessary interactions

For Example,

- Traditional Unix has terrible compartmentalization
  - Obtaining root privileges gives away the entire system
- Redesigns that allow root programs to run under other identities help
  - E.g., mail server and print server users
- Not just a problem for root programs
  - E.g., web servers that work for many clients
- Research systems like Asbestos allow finer granularity compartmentalization

6. Value Simplicity

- Complexity is the enemy of security
- Complex systems give more opportunities to screw up
- Also, harder to understand all “proper” behaviors of complex systems
- So favor simple designs over complex ones

For Example,

- Re-use components when you think they’re secure
- Use one implementation of encryption, not several
  - Especially if you use “tried and true” implementation
- Build code that only does what you need
  - Implementation of exactly what you need are safer than “Swiss army knife” approaches
- Choose simple algorithms over complex algorithms
  - Unless complex one offers *necessary* advantages
  - “It’s somewhat faster” usually isn’t a necessary advantage
  - And “it’s a neat new approach” definitely isn’t

Especially Important When Human Users Involved

- Users will not read documentation
  - So don’t rely on designs that require that
- Users are lazy
  - They’ll ignore pop-ups and warnings
  - They will prioritize getting the job done over security
  - So designs requiring complex user decisions usually fail

7. Promote Privacy

- Avoid doing things that will compromise user privacy
- Don’t ask for data you don’t need
- Avoid storing user data permanently
  - Especially unencrypted data
- There are strong legal issues related to this, nowadays

For Example,

- Google’s little war driving incident
- They drove around many parts of the world to get information on Wifi hotspots
- But they simultaneously were sniffing and storing packets from those networks
- And gathered a lot of private information
- They got into a good deal of trouble . . .

8. Remember That Hiding Secrets is Hard

- Assume anyone who has your program can learn <u>everything</u> about it
- "Hidden" keys, passwords, certificates in executables are invariably found
- Security based on obfusticated code is always broken
- Just because you're not smart enough to crack it doesn't mean the hacker isn't, either

For Example,

- Passwords often “hidden” in executables
  - Zhuhai RaySharp surveillance DVRs had one hard coded password on all 46,000+
  - Allowed Internet access to any of them
- Android apps containing private keys are in use (and are compromised)
- Ubiquitous in digital rights management
  - And it never works

9. Be Reluctant to Trust

- Don’t automatically trust things
  - Especially if you don’t have to
- Remember, you’re not just trusting the honesty of the other party
  - You’re also trusting their caution
- Avoid trusting users you don’t need to trust, too
  - Doing so makes you more open to social engineering attacks

For Example,

- Why do you trust that commercial software?
- Or that open source library?
- Must you?
- Can you design the system so it’s secure even if that component fails?
- If so, do it

10. Use Your Community Resources

- Favor widely used and respected security software over untested stuff
  - Especially your own . . .
- Keep up to date on what’s going on
  - Not just patching
  - Also things like attack trends

For Example,

- Don’t implement your own AES code
- Rely on one of the widely used versions
- But also don’t be too trusting
  - E.g., just because it’s open source doesn’t mean it’s more secure
  - We’ll discuss openssl and Heartbleed

Choosing Technologies

- Different technologies have different security properties
  - Operating systems
  - Languages
  - Object management systems
  - Libraries
- Important to choose wisely
  - Understand the implications of the choice

Choices and Practicalities

- You usually don’t get to choose the OS
- The environment you’re writing for dictates the choice
  - E.g., commercial software often must be written for Windows
  - Or Linux is the platform in your company
- Might not get choice in other areas, either
  - But exercise it when you can

Operating System Choices

- Rarely an option, and does it matter anyway?
- Probably not, any more
  - All major choices have poor security histories
    - No, Linux is not necessarily safer than Windows
  - All have exhibited lots of problems
  - In many cases, problems are in the apps, anyway
- Exception if you get to choose a really trusted platform
  - E.g., SE Linux
    - Not perfect, but better
    - At a cost in various dimensions

Language Choices

- More likely to be possible
  - Though often hard to switch from what’s already being used
- If you do get the choice, what should it be?

C and C++

- Probably the worst security choice
- Far more susceptible to buffer overflows than other choices
- Also prone to other reliability problems
- Often chosen for efficiency
  - But is efficiency that important for your application?

Java

- Less susceptible to buffer overflows
- Also better error handling than C/C++
- Has special built-in security features
  - Which aren't widely used
- But has its own set of problems
  - E.g., exception handling issues
  - And issues of inheritance
- 9 remotely exploitable security flaws in 2nd quarter 2016 Oracle patches
- Multiple serious security problems in recent years

Python

- A type-safe language
  - So buffer overflows usually not possible
- But many extensibility features
  - YAML, pickles, import paths
  - Attackers can use these to get malicious code into your program
- Injection attacks (SQL, commands)

Scripting Languages

- Depends on language
- Many are type safe (or non-typed), limiting buffer overflow possibilities
- Javascript and CGIbin have awful security reputations
- Perl offers some useful security features
- But there are some general issues

Scripting Language Security Issues

- Might be security flaws in their interpreters
  - More likely than in compilers
- Scripts often easily examined by attackers
  - Obscurity of binary is no guarantee, but it is an obstacle
- Scripting languages often used to make system calls
  - Inherently dangerous, esp. things like `eval()`
- If they call libraries, there can be overflows there
  - E.g., Python buffer overflow in 2014
- Many script programmers don’t think about security at all

Open Source vs. Closed Source

- Some argue open source software is inherently more secure
- The "many eyes" argument –
  - Since anyone can look at open source code,
  - More people will examine it
  - Finding more bugs
  - Increasing security

Is the "Many Eyes" Argument Correct?

- Probably not
- At least not in general
- Linux has security bug history similar to Windows
- Other open source projects are even worse
  - In many cases, nobody really looks at the code
  - Which is no better than closed source

The Flip Side Argument

- “Hackers can examine open source software and find its flaws”
- Well, Windows’ security history is not a recommendation for this view
- Most commonly exploited flaws can be found via black-box approach
  - E.g., typical buffer overflows

The Upshot?

- No solid evidence that open source or closed source produces better security
- Major exception is crypto
  - At least for crypto standards
  - Maybe widely used crypto packages
  - Criticality and limited scope means many eyeballs will really look at it

One More Consideration

- The Snowden leaks suggest some companies put trapdoors in software
  - Especially security-related software
- When it's closed source, nobody else can check that
- When it's open source, maybe they can
  - Emphasis on the "maybe," though

Major Problem Areas for Secure Programming

- Certain areas of programming have proven to be particularly prone to problems
- What are they?
- How do you avoid falling into these traps?

Example Problem Areas

- Buffer overflows and other input verification issues
- Error handling
- Privilege escalation
- Race conditions
- Use of randomness
- Proper use of cryptography
- Trust
- Variable synchronization
- Variable initialization
- There are others . . .

Buffer Overflows

- The poster child of insecure programming
- One of the most commonly exploited types of programming error
- Technical details of how they occur discussed earlier
- Key problem is language does not check bounds of variables

Preventing Buffer Overflows

- Use a language with bounds checking
  - Most modern languages other than C and C++ (and assembler)
  - Not always a choice, or the right choice
  - Not always entirely free of overflows
- Use compiler options that perform bounds checking
- Check bounds carefully yourself
- Avoid constructs that often cause trouble

Problematic Constructs for Buffer Overflows

- Most frequently C library calls:
  - `gets(), strcpy(), strcat(), sprintf(), scanf(), sscanf(), fscanf(), vfscanf(),vsprintf(), vscanf(), vsscanf(), streadd(), strecpy()`
  - There are others that are also risky

Why Are These Calls Risky?

- They copy data into a buffer
- Without checking if the length of the data copied is greater than the buffer
- Allowing overflow of that buffer
- Assumes attacker can put his own data into the buffer
  - Not always true
  - But why take the risk?

What Do You Do Instead?

- Many of the calls have variants that specify how much data is copied
  - If used properly, won’t allow the buffer to overflow
- Those without the variants allow precision specifiers
  - Which limit the amount of data handled

Is That All I Have To Do?

- No
- These are automated buffer overflows
- You can easily write your own
- Must carefully check the amount of data you copy if you do
- And beware of integer overflow problems

An Example

- Actual bug in OpenSSH server:

```
u_int nresp;
. . .
nresp = packet_get_int();
If (nresp > 0) {
  response = xmalloc(nresp * sizeof(char *));
  for (i=0; i<nresp;i++)
      response[i] = packet_get_string(NULL);
}
packet_check_eom();
```

Why Is This a Problem?

- `nresp` is provided by the user
  - `nresp = packet_get_int();`
- But we allocate a buffer of `nresp` entries, right?
  - `response = xmalloc(nresp * sizeof(char *));`
- So how can that buffer overflow?
- Due to integer overflow

How Does That Work?

- The argument to `xmalloc()` is an unsigned int
- Its maximum value is 2^{32}-1
  - 4,294,967,295
- `sizeof(char *)` is 4
- What if the user sets `nresp` to 0x40000020?
- Multiplication is modulo 2^{32} . . .
  - So 4 * 0x40000020 is 0x80

What Is the Result?

- There are 128 entries in `response[]`
- And the loop iterates hundreds of millions of times
  - Copying data into the "proper place" in the buffer each time
- A massive buffer overflow

Other Programming Tools for Buffer Overflow Prevention

- Software scanning tools that look for buffer overflows
  - Of varying sophistication
- Use a C/C++ compiler that includes bounds checking
  - Most do, typically offered as an option
- Use integrity-checking programs
  - Stackguard, Rational’s Purity, etc.

Canary Values

- One method of detecting buffer overflows
- Akin to the “canary in the mine”
- Place random value at end of data structure
- If value is not there later, buffer overflow might have occurred
- Implemented in language or OS

Data Execution Prevention (DEP)

- Buffer overflows typically write executable code somewhere
- DEP prevents this
  - Page is either writable or executable
- So if overflow can write somewhere, can’t execute the code
- Present in Windows, Mac OS, etc.
- Doesn’t help against some advanced techniques

Randomizing Address Space (ASLR)

- Address Space Layout Randomization
- Randomly move around where things are stored
  - Base address, libraries, heaps, stack
- Making it hard for attacker to write working overflow code
- Used in Windows, Linux, MacOS
- Not always used, not totally effective
  - Several recent Windows problems from programs not using ASLR

Other Input Verification Problems

- Even if you don't have buffer overflows, you can get dangerous input
- You need to verify that the input you get is safe to use
- Many problems arise from not validating input
  - Or allowing arbitrary input

The Heartbleed Bug

- A bug in the OpenSSL implementation of SSL/TLS
- Effect was to reveal potentially secret information from a remote process
  - Cryptographic keys, passwords, Social Security Numbers, etc.
- A *buffer overread* problem

SSL Heartbeat Messages

- SSL requires heartbeat messages
- One side sends the other a buffer containing some content
- The other side must send the same content back
- The heartbeat message contains the buffer and its length

The Security Flaw

- The OpenSSL implementation did not verify the length field
  - Didn’t check that it matched the actual buffer supplied
- Instead, it returned whatever was in the buffer plus any extra specified length
- Which was something else in the process’ memory

For Example,

- The attacker sends a buffer containing “hello” and a buffer length of 512
- The other side of the SSL connection returns “hello”
  - Plus the next 507 characters in its memory
  - Which might contain “interesting” data

Exploiting Heartbleed

- Establish an SSL connection to a vulnerable server
- Repeatedly send these buffer overread requests
- Examine what you get back looking for “good stuff”

Fixing Heartbleed

- Simple in principle
- Update the software to check the buffer length in heartbeat messages
- In practice, required updating software in literally millions of sites
  - Some still not patched

Impacts of Heartbleed

- Could be used to steal private keys associated with certificates
  - Many web sites had to get new certs
  - Many that should haven't
  - Old certs are still out there
- One Heartbleed attack compromised 4.5 million hospital patient records