Designing and Building Secure Software (Part 2)

With material from Dave Levin, Mike Hicks
• Tuesday = Exam review day
  • Bring your questions or class will be short
• Thursday = Exam day
  • One side of one sheet
Last time

- Threat modeling
- Security requirements
- Beginning of design principles
Some Principles

• Favor simplicity
  - Use fail-safe defaults
  - Do not expect expert users

• Trust with reluctance
  - Minimize trusted computing base
  - Grant the least privilege possible; compartmentalize

• Defend in Depth
  - If one fails, maybe the next will succeed
  - Use community resources to stack defenses

• Monitor and trace
Rule: Input validation

• Input validation is a kind of least privilege
  • Trust a subsystem only **under certain circumstances**
    • Validate that those circumstances hold

• Several **examples** so far:
  • Trust a given function *if* the range of its parameters is limited (e.g., within the length of a buffer)
  • Trust a client form field *if* it contains no `<script>` tags (and other code-interpretable strings)
  • Trust a YAML-encoded string *if* it contains no code
TwR: Compartmentalization

- **Isolate** a component in a compartment or sandbox
  - Reduced privilege: Certain interactions impossible
- **Category**: Prevention and Mitigation
- **Example**: Disconnect student records database from the Internet
  - Grant access only by direct terminals
- **Example**: `seccomp` system call in Linux (next)
SecComp

• Linux system call enabled since 2.6.12 (2005)
  • Affected process can subsequently only perform read, write, exit, and sigreturn system calls
  • No support for open call: Only use already-open files
  • Isolates a process by limiting possible interactions

• Follow-on work produced seccomp-bpf
  • Limit process to policy-specific set of system calls, subject to a policy handled by the kernel
  • Used by Chrome, OpenSSH, vsftpd, and others
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call fork to create a new process
- In the new process, open the file
- Call exec to run Flash player
- Call seccomp-bpf to compartmentalize
TwR: Promote Privacy

• Goal: restrict flow of sensitive data as much as possible
  • Another version of compartmentalization

• Category: Mitigation

• Example: A student admission system receives (sensitive) letters of recommendation as PDF files
  • A typical design would allow reviewers to download these files for viewing on their local computers
  • But then compromise of these computers leaks private information
  • Better: PDFs only viewable in browser; no data downloaded to client machine.
Some Principles

• Favor simplicity
  • Use fail-safe defaults
  • Do not expect expert users

• Trust with reluctance
  • Minimize trusted computing base
  • Grant the least privilege possible; compartmentalize

• Defend in Depth
  • If one fails, maybe the next will succeed
  • Use community resources to stack defenses

• Monitor and trace
Defense in Depth (DiD)

- Security by diversity
  - If one layer is broken, another must be bypassed
  - Must be materially different
- Categories: Prevention/Mitigation

- Example: Do all of the following, not just one
  - Use a firewall for preventing access via non-web ports
  - Encrypt account data at rest
  - Use a safe language to avoid low-level vulnerabilities
Failure: Authentication Bypass

- Poor passwords can be **guessed**
  - Bypassing authentication intent
- Passwords can be **stolen**
  - **Defense in depth**: Hash and salt password database, use a slow hash
  - Assumes compromise is possible, uses second-level defense
DiD: Use community resources

- Use hardened code, perhaps from other projects
  - E.g., crypto libraries
  - But make sure it meets your needs (*test it; cf. Heartbleed!*)

- Vet designs **publicly**: No security by obscurity!

- Stay up on recent threats and research
  - NIST for standards
  - OWASP, CERT, Bugtraq for vulnerability reports
  - SANS Newsbites for latest top threats
  - Academic and industry conferences and journals for longer term trends, technology, and risks
  - etc.
Failure: Broken Crypto Impl.

Getting crypto right is hard

Remote Timing Attacks are Practical

David Brumley
Stanford University
dbrumley@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Abstract

Timing attacks are usually used to attack weak computing devices such as smartcards. We show that timing attacks apply to general software systems. Specifically, we devise a timing attack against OpenSSL. Our experiments show that we can extract private keys from an OpenSSL-based web server running on a machine in the local network. Our results demonstrate that timing attacks against network servers are practical and therefore security systems should defend against them.

The attacking machine and the server were in different buildings with three routers and multiple switches between them. With this setup we were able to extract the SSL private key from common SSL applications such as a web server (Apache+mod_SSL) and a SSL-tunnel.

Interprocess. We successfully mounted the attack between two processes running on the same machine. A hosting center that hosts two domains on the same machine might give management access to the admins of each domain. Since both domain are hosted on the same machine, one admin could use the attack to extract the secret key belonging to the other domain.

USENIX Security’03

Timing channel: During decryption (chosen ciphertext)
Failure: Broken Crypto Impl.

Getting crypto right is hard

Use vetted implementations and algorithms

Widespread Weak Keys in Network Devices

We performed a large-scale study of RSA and DSA cryptographic keys in use on the Internet and discovered that significant numbers of keys are insecure due to insufficient randomness. These keys are being used to secure TLS (HTTPS) and SSH connections for hundreds of thousands of hosts.

- We found that 5.57% of TLS hosts and 9.60% of SSH hosts share public keys in an apparently vulnerable manner, due to either insufficient

Q. Why are repeated public keys a problem?

If two servers share the same public key, that means that they also share the same private key, and thus have the ability to decrypt messages to or to impersonate the other. In many cases, we observed the same public key being served from thousands of independent hosts; if the private key is ever compromised on any one of them, all of the hosts' keys will have been compromised.

Q. Why are default keys a problem?

Many kinds of network devices "bake in" a default key which is used on every instance of a particular model. Default keys pose the same security risks as repeated keys; in addition, it may be possible for an attacker to extract the private keys from the device firmware and be able to compromise every model of that device.

Poor randomness leads to repeated keys.
Monitoring and Traceability

• If you are attacked, **how will you know?**
  • Once you learn, **how will you find the cause?**

• **Software must log** relevant operational information
  • Logs must be **append-only** (best = use crypto)!
  • What to log? E.g., events handled, packets processed, requests satisfied, …

  • **Category:** Detection and Recovery

• **Log aggregation:** Correlate activities of multiple applications when diagnosing a breach
  • E.g., [splunk log aggregator](https://www.splunk.com)

It’s the most impenetrable lock on the market today. It has only one design flaw: the door must be closed!
Top 10 Flaws: Do not ...

1. Assume trust, rather than explicitly give it or award it
2. Use an authentication mechanism that can be bypassed, tampered with
3. Authorize without considering sufficient context
4. Confuse data and control instructions, or process control instructions from untrusted sources
5. Fail to validate data *explicitly* and *comprehensively*
6. Fail to use cryptography correctly
7. Fail to identify sensitive data and how to handle it
8. Ignore the users
9. Integrate external components without considering their attack surface
10. Rigidly constrain future changes to objects and actors

http://cybersecurity.ieee.org/center-for-secure-design.html
Top 10 Flaws: Do not …

1. Assume trust, rather than explicitly give it or award it
2. Use an authentication mechanism that can be bypassed or tampered with
3. Authorize without considering sufficient context
4. Confuse data and control instructions, and process control instructions from untrusted sources
5. Fail to validate data explicitly and comprehensively
6. Fail to use cryptography correctly
7. Fail to identify sensitive data and how to handle it
8. Ignore the users
9. Integrate external components without considering their attack surface
10. Rigidly constrain future changes to objects and actors

Several of these we’ve covered already; will consider several in more detail

http://cybersecurity.ieee.org/center-for-secure-design.html
Failure: Authentication Bypass

- Client accepts invalid SSL certificates
  - Am I really talking to my bank, or a site pretending to be my bank?

- Web browser warning
  - But how many users will click right through?
Failure: Authentication Bypass

- **Mobile apps use SSL behind the scenes**: what happens when an app receives an invalid certificate?
  - “While it is understandable that developers **turn off SSL certificate validation** in the development phase, these developers basically forgot to remove their accept-all code when they released their apps.”

- **Remember**: Security is not a feature
  - Need to test what should *not* happen
Failure: Authentication Bypass

- **Authentication tokens with long timeouts**
  - Motivates brute-force attempts to steal session cookies
  - Recall Twitter auth_token failure from web security
  - But can’t make it too short, or will irritate users

- **In general**: avoid authentication bypass by developing good abuse cases, violating assumption of unique knowledge or possession
  - How might an adversary learn a password? Spoof a biometric? Steal a session ID?
Failure: Bad (or Wrong) Crypto

• (I repeat) **Don’t roll your own crypto**
  • Both design and implementation are hard to get right

• **Don’t assume it gives you something it doesn’t**
  • Encryption may protect confidentiality but not integrity. Hashing protects integrity but not confidentiality.

• **Know how to use it properly**
  • Use *properly generated keys* of *sufficient size*
  • **Protect the keys** from compromise
    • Don’t hard-code them, or embed them in released binaries
Example: Mobile Banking

- Used in developing world to transfer $ via phone
  - Examined 7 major vendors
- Four failed to validate certificates properly
- Four used non-standard crypto
- All but one had some major flaw

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/reaves
• Example: Money on Mobile (India)

• All messages sent via plaintext HTTP

• This is the only crypto used in the app!
Think about data sources: Which require protection?

- Personally identifiable information, sensor readings, cryptographic keys, session tokens, geolocation data, …
  - Failure: private data exposed to general access

How are these data sources exposed?

- When at rest, when in transmission … what is the threat model?
  - Failure: embedding authentication token in exposed URL

How does data, and its exposure, change as the application evolves over time?

Failure: Ignore which data is sensitive
Case study: VSFTPD
Secure coding in C

• Since the language provides few guarantees, developers must use discipline

• Good reference guide: CERT C Coding Standard
  • https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Coding+Standard
    • Similar guides for other languages (e.g., Java)
  • See also: David Wheeler: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/internals.html

Combine with advanced code review and testing
Very Secure FTPD

- **FTP**: File Transfer Protocol
  - More popular earlier, but still in use
  - 90’s and 00’s: **FTP daemon compromises frequent and costly** (Wu-FTPD, ProFTPD, …)

- **Thoughtful design** and **implementation**
  - To **prevent** and **mitigate** flaws
  - But also to achieve **good performance**
  - Written in C

- No known breaches (since 2002)

[https://security.appspot.com/vsftpd.html](https://security.appspot.com/vsftpd.html)
VSFTPD Threat model

- Clients untrusted until authenticated
- Once authenticated, limited trust:
  - According to user’s file access control policy
  - For the files being served via FTP (and not others)
- Possible attack goals
  - Steal or corrupt resources (e.g., files, malware)
  - Remote code injection
- Scenarios: Client attacks server or another client
Insecure design (other FTPDs)

- One process per user, runs as root
  - Any vulnerability (buffer, format string) leads to **root compromise**
  - Even for anonymous users!

- **Better:** Separate functions across processes with different privileges; use interprocess comms
Defensive design: Minimal privilege

- **Untrusted input** always handled by **non-root process**
  - Uses IPC to delegate high-privilege actions
  - Very little code runs as **root**

- **Reduce privileges** as much as possible
  - Run as particular (unprivileged) user
    - File system access control enforced by OS
    - Use capabilities and/or SecComp on Linux
      - Reduces the system calls a process can make

- **chroot to hide all directories** but the current one
  - Keeps visible only those files served by FTP
Connection Establishment
Performing Commands

connection server
command processor
CHOWN
OK

command reader/executor
CHDIR
OK

client
Defense: Login reader

- **Whitelist** input
  - Allowed input very limited
  - Limits attack surface

- **Limited** privilege
  - Not root; authentication is separate
  - Mutes damage of injected code

- Command processor *only talks to* reader
  - Again, whitelist limited input
Defense: Commands

- Command reader **sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege

- Command processor **only talks to** reader
  - Again, whitelist limited input
Attack: Cross-session
Attack: Cross-session

- Each session isolated
- Can only talk to one client

connection server

command reader/executor

CMD

ATTACK

client 1

CMD

client 2
Secure C implementation in vsftpd
Defense: Secure Strings

```c
struct mystr {
    char* PRIVATE_HANDS_OFF_p_buf;
    unsigned int PRIVATE_HANDS_OFF_len;
    unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Normal (zero-terminated) C string

Actual length (i.e., `strlen(PRIVATE_HANDS_OFF_p_buf)`) 

Size of buffer returned by `malloc`
void
private_str_alloc_memchunk(struct mystr* p_str,  
const char* p_src, unsigned int len)
{
    ...
}

void
str_copy(struct mystr* p_dest, const struct mystr* p_src)
{
    private_str_alloc_memchunk(p_dest, p_src->p_buf,  
                               p_src->len);
}

struct mystr
{
    char* p_buf;
    unsigned int len;
    unsigned int alloc_bytes;
};

replace uses of char* with struct mystr*  
and uses of strcpy with str_copy
void private_str_alloc_memchunk(struct mystr* p_str, const char* p_src, unsigned int len)
{
    /* Make sure this will fit in the buffer */
    unsigned int buf_needed;
    if (len + 1 < len)
    {
        bug("integer overflow");
    }
    buf_needed = len + 1;
    if (buf_needed > p_str->alloc_bytes)
    {
        str_free(p_str);
        s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
        p_str->alloc_bytes = buf_needed;
    }
    vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
    p_str->p_buf[len] = '\0';
    p_str->len = len;
}
Defense: Secure Stdcalls

• Common problem: error handling
  • Libraries assume that arguments are well-formed
  • Clients assume that library calls always succeed

• Example: malloc()
  • What if argument is non-positive?
    • We saw earlier that integer overflows can induce this behavior
    • Leads to buffer overruns
  • What if returned value is NULL?
    • Oftentimes, a dereference means a crash
    • If no memory protection, can cause corruption
void* vsf_sysutil_malloc(unsigned int size)
{
    void* p_ret;
    /* Paranoia - what if we got an integer overflow/underflow? */
    if (size == 0 || size > INT_MAX)
    {
        bug("zero or big size in vsf_sysutil_malloc");
    }
    p_ret = malloc(size);
    if (p_ret == NULL)
    {
        die("malloc");
    }
    return p_ret;
}