Security Requirements

• **Software requirements** typically about **what** the software should do

• We also want to have **security requirements**
  - **Security-related goals** *(or policies)*
    - Example: One user's bank account balance should not be learned by, or modified by, another user, unless authorized
  - **Required mechanisms for enforcing them**
    - Example:
      1. Users identify themselves using passwords,
      2. Passwords must be "strong," and
      3. The password database is only accessible to login program.
Typical *Kinds* of Requirements

- **Policies**
  - **Confidentiality** (and Privacy and Anonymity)
  - **Integrity**
  - **Availability**

- **Supporting mechanisms**
  - **Authentication**
  - **Authorization**
  - **Audit-ability**

Supporting mechanisms

These relate identities ("principals") to actions

<table>
<thead>
<tr>
<th><strong>Authentication</strong></th>
<th><strong>Authorization</strong></th>
<th><strong>Audit-ability</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>How can a system tell <em>who a user is</em></td>
<td>How can a system tell <em>what a user is allowed to do</em></td>
<td>How can a system tell <em>what a user did</em></td>
</tr>
<tr>
<td>What we know</td>
<td>Access control policies (defines) + <em>Mediator</em> (checks)</td>
<td>Retain enough info to determine the circumstances of a breach</td>
</tr>
<tr>
<td>What we have</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What we are</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;1 of the above = <em>Multi-factor authentication</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Privacy and Confidentiality

• Definition: Sensitive information not leaked to unauthorized parties
• Broadly speaking: privacy for individuals, confidentiality for data

• Example policy: bank account status (including balance) known only to the account owner

• Leaking directly or via side channels
  • Example: manipulating the system to directly display Bob’s bank balance to Alice
  • Example: determining Bob has an account at Bank A according to shorter delay on login failure

Secrecy vs. Privacy? https://www.youtube.com/watch?v=Nil7 YM71k5U

Anonymity

• A specific kind of privacy

• Example: Non-account holders should be able to browse the bank informational site without being tracked
  • Here the adversary is the bank
  • The previous examples considered other account holders as possible adversaries
Integrity

• Definition: Sensitive information not damaged by (computations acting on behalf of) unauthorized parties

• Example: Only the account owner can authorize withdrawals from her account

• Violations of integrity can also be direct or indirect
  • Example: Being able specifically withdraw from the account vs. confusing the system into doing it

Availability

• Definition: A system is responsive to requests

• Example: a user may always access her account for balance queries or withdrawals

• Denial of Service (DoS) attacks attempt to compromise availability
  • by busying a system with useless work
  • or cutting off network access
Supporting mechanisms

- Leslie Lamport's **Gold Standard** defines mechanisms provided by a system to enforce its requirements
  - **Authentication**
  - **Authorization**
  - **Audit**

- The gold standard is **both requirement and design**
  - The **sorts of policies** that are authorized **determines** the **authorization mechanism**
  - The **sorts of users** a system has **determines** how they should be **authenticated**

---

Authentication

- What is the **subject of security policies**?
  - Need to define a **notion of identity** ("principal") and a way to **connect an action with an identity**

- **How can system tell a user is who he says he is?**
  - What we **know** (e.g., password)
  - What we **are** (e.g., fingerprint)
  - What we **have** (e.g., key)
  - Authentication mechanisms that employ more than one of these factors are called **multi-factor authentication**
Authorization

• Defines **when a principal may perform an action**

• **Example**: Bob is authorized to access his own account, but not Alice's account

• There are a wide variety of **policies** that define what actions might be authorized
  • E.g., access control policies, which could be originator based, role-based, user-based, etc.

Audit

• Retain enough information to be able to **determine the circumstances of a breach or misbehavior** (or **establish one did not occur**)
  • Such information, often stored in **log files**, must be **protected from tampering**, and from access that might violate other policies

• **Example**: Every account-related action is logged locally and mirrored at a separate site

• How long should you **retain** logs?
Top Design Flaws

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
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

Failure: Authentication Bypass

- **Clients coerced to accept invalid SSL certificates**
- **Bypasses client authentication of server:**
  Am I really talking to my bank, or a site pretending to be my bank?
- **Web browser presents a warning**
  - But how many users will “click through?”
**Failure: Authentication Bypass**

- **Mobile apps use SSL behind the scenes**: what happens when an app gets 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

---

**Authentication Bypass**

- **Authentication tokens with long timeouts**
  - Motivates brute-force attempts to steal session cookies
    - Recall Twitter auth_token failure from web security unit
  - 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**
  - Per use-community-resources examples: both design and implementation are hard to get right

- **Don’t assume it gives you something it doesn’t**
  - Encryption algorithm 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

Failure: Ignore which data is sensitive

- **Think carefully 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 Attack Surface of External Components

- **Attack surface**: Elements of a system that an adversary can attack, or use in an attack

- **Do third-party components do only what I want?**
  - **Shellshock**: Failure: “Bourne again shell” (bash) — used by web sites (for CGI) DHCP, and other functions — is far more powerful than necessary for these tasks. Thus: failure in bash leads to a serious network-facing vulnerability.

---

Case study: VSFTPD
Very Secure FTPD

- **FTP**: File Transfer Protocol
  - More popular before the rise of HTTP, but still in use
  - 90's and 00's: FTP daemon compromises were frequent and costly, e.g., in Wu-FTPD, ProFTPd, …

- Very thoughtful design aimed to prevent and mitigate security defects

- But also to achieve good performance
  - Written in C

- Written and maintained by Chris Evans since 2002
  - No security breaches that I know of
    
    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 FTP (and not others)

- Possible attack goals
  - Steal or corrupt resources (e.g., files, malware)
  - Remote code injection

- Circumstances:
  - Client attacks server
  - Client attacks another client
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
The actual length (i.e., `strlen(PRIVATE_HANDS_OFF_p_buf)`) Size of buffer returned by `malloc`

```c
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);
}
```

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;
}

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

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
    • On platforms without memory protection, a dereference can cause corruption
```c
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;
}
```

Defense: 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
Logging out

Attack: Login

- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input
Attack: Commands

- Command reader sandboxed
  - Not root
  - Handles most commands
  - Except few requiring privilege
- Comm. proc. only talks to reader
  - And, again, white-lists its limited input

Attack: Cross-session
**Attack: Cross-session**

- Each session isolated
- Only can talk to one client

---

**Separation of responsibilities**

**TCB: KISS**

**TCB: Privilege separation**

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as simple as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distributed. Here are example requests:

   - chown() request. The child may request a recently uploaded file get chown('ed') to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. All OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing: it’s actually quite hard to do, but vsftpd manages it in the name of being secure. I’m unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

---

**Kerckhoff’s principle!**