Making secure software

- Flawed approach: Design and build software, and ignore security at first
  - Add security once the functional requirements are satisfied
- Better approach: Build security in from the start
  - Incorporate security-minded thinking into all phases of the development process

Threat Model

- The threat model makes explicit the adversary’s assumed powers
  - Consequence: The threat model must match reality, otherwise the risk analysis of the system will be wrong
- The threat model is critically important
  - If you are not explicit about what the attacker can do, how can you assess whether your design will repel that attacker?
  - This is part of architectural risk analysis
Example: **Network User**

- An (anonymous) user that can connect to a service via the network
- Can:
  - measure the size and timing of requests and responses
  - run **parallel sessions**
  - provide **malformed inputs, malformed messages**
  - drop or send extra messages
- **Example attacks:** SQL injection, XSS, CSRF, buffer overrun/ROP payloads, …

---

Example: **Snooping User**

- Internet user **on the same network** as other users of some service
  - For example, someone connected to an unencrypted Wi-Fi network at a coffee shop
- Thus, can additionally
  - Read/measure others’ messages,
  - Intercept, duplicate, and **modify messages**
- **Example attacks:** Session hijacking (and other data theft), privacy-violating side-channel attack, denial of service

---

Example: **Co-located User**

- Internet user **on the same machine** as other users of some service
  - E.g., **malware** installed on a user’s laptop
- Thus, can additionally
  - Read/write user’s files (e.g., cookies) and memory
  - Snoop keypresses and other events
  - Read/write the user’s display (e.g., to spoof)
- **Example attacks:** Password theft (and other credentials/secrets)

---

**Threat-driven Design**

- Different threat models will elicit different responses
- **Network-only attackers** implies message traffic is safe
  - No need to encrypt communications
  - This is what telnet remote login software assumed
- **Snooping attackers** means message traffic is visible
  - So use encrypted wifi (link layer), encrypted network layer (IPsec), or encrypted application layer (SSL)
  - Which is most appropriate for your system?
- **Co-located attacker** can access local files, memory
  - Cannot store unencrypted secrets, like passwords
Bad Model = Bad Security

- Any assumptions you make in your model are potential holes that the adversary can exploit
- E.g.: Assuming no snooping users no longer valid
- Prevalence of wi-fi networks in most deployments
- Other mistaken assumptions
  - **Assumption:** Encrypted traffic carries no information
    - Not true! By analyzing the size and distribution of messages, you can infer application state
  - **Assumption:** Timing channels carry little information
    - Not true! Timing measurements of previous RSA implementations could be used eventually reveal a remote SSL secret key

Finding a good model

- **Compare against similar systems**
  - What attacks does their design contend with?

- **Understand past attacks and attack patterns**
  - How do they apply to your system?

- **Challenge assumptions in your design**
  - What happens if an assumption is untrue?
    - What would a breach potentially cost you?
  - How hard would it be to get rid of an assumption, allowing for a stronger adversary?
    - What would that development cost?

Security Requirements

Running Example: **On-line banking**
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**
  - **Auditability**

Privacy and Confidentiality

- **Definition**: *Sensitive information not leaked* to unauthorized parties
- Called *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

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

Secrecy vs. Privacy? [https://www.youtube.com/watch?v=NIJ7YM71kSU](https://www.youtube.com/watch?v=NIJ7YM71kSU)
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 and a way to connect an action with an identity
    - a.k.a. a principal
- **How can system tell a user is who he says he is?**
  - What (only) he knows (e.g., password)
  - What he is (e.g., biometric)
  - What he has (e.g., smartphone)
  - Authentication mechanisms that employ more than one of these factors are called multi-factor authentication
    - E.g., bank may employ passwords and text of a special code to a user’s smart phone
## 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

## Defining Security Requirements

- Many processes for deciding security requirements
- **Example**: **General policy concerns**
  - Due to **regulations/standards** (HIPAA, SOX, etc.)
  - Due **organizational values** (e.g., valuing privacy)
- **Example**: **Policy arising from threat modeling**
  - Which **attacks** cause the **greatest concern**?
    - Who are the likely adversaries and what are their goals and methods?
  - Which **attacks** have **already occurred**?
    - Within the organization, or elsewhere on related systems?

## Abuse Cases

- Abuse cases illustrate security requirements
- Where use cases describe what a system **should do**, **abuse cases describe what it should not do**
- **Example use case**: The system allows bank managers to modify an account’s interest rate
- **Example abuse case**: A user is able to spoof being a manager and thereby change the interest rate on an account
Defining Abuse Cases

- Using attack patterns and likely scenarios, construct cases in which an adversary’s exercise of power could violate a security requirement
  - Based on the threat model
  - What might occur if a security measure was removed?

  **Example**: Co-located attacker steals password file and learns all user passwords
  - Possible if password file is not encrypted

  **Example**: Snooping attacker replays a captured message, effecting a bank withdrawal
  - Possible if messages are have no nonce

Design Flaws

Design Defects = Flaws

- Recall that software defects consist of both flaws and bugs
  - **Flaws** are problems in the design
  - **Bugs** are problems in the implementation

- We avoid flaws during the design phase

  According to Gary McGraw, 50% of security problems are flaws
  - So this phase is very important

Design vs. Implementation?

- Many different levels of system design decisions
  - **Highest level**: main actors (processes), interactions, and programming language(s) to use

  **Next level**: decomposition of an actor into modules/components, identifying the core functionalities and how they work together

  **Next level**: how to implement data types and functions, e.g., purely functionally, or using parallelism, etc.

- Last two could be implementation or design, or both
  - The distinction is a bit fuzzy
Secure Software Design

- **Design** software architecture according to good principles and rules
- **Risk-based analysis** of software architecture’s design

Principles and Rules

- A **principle** is a high-level design goal with many possible manifestations
- A **rule** is a specific practice that is consonant with sound design principles
  - The **difference between these two can be fuzzy**, just as design vs. implementation is fuzzy.
    - For example, there is often a principle underlying specific practices
  - Principles often overlap
- The **software design phase** tends to **focus on principles** for avoiding flaws

Categories of Principles

- **Prevention**
  - **Goal**: Eliminate software defects entirely
  - **Example**: Heartbleed bug would have been prevented by using a type-safe language, like Java
- **Mitigation**
  - **Goal**: Reduce the harm from exploitation of unknown defects
  - **Example**: Run each browser tab in a separate process, so exploitation of one tab does not yield access to data in another
- **Detection** (and **Recovery**)
  - **Goal**: Identify and understand an attack (and undo damage)
  - **Example**: Monitoring (e.g., expected invariants), snapshotting

The Principles

- **Favor simplicity**
  - Use fail-safe defaults
  - Do not expect expert users
- **Trust with reluctance**
  - Employ a small trusted computing base
  - Grant the least privilege possible
    - Promote privacy
    - Compartmentalize
- **Defend in Depth**
  - Use community resources - no security by obscurity
- **Monitor and trace**
Classic Advice

The classic reference on principles of secure design is *The Protection of Information in Computer Systems*, by Saltzer and Schroeder (in 1975)

**Principles**
- Economy of Mechanism
- Fail-safe Defaults
- Complete mediation
- Open design
- Psychological acceptability
- Separation of privilege
- Least privilege
- Least common mechanism
- (Work factor)
- (Compromise recording)

Comparing to our list

- Several principles reorganized/renamed
  - *Separation of privilege* has elements of our *compartmentalization, defend in depth*
  - *Open design* is like *use community resources*, but did not anticipate open-source code
- Monitoring is added
  - Their focus on prevention of attack, rather than recovery
- “Principle” of *complete mediation* dropped
  - CM not a *design* principle, but a rather an implementation requirement

Design Category: Favor Simplicity

**Favor Simplicity**

- Keep it *so simple it is obviously correct*
  - Applies to the external interface, the internal design, and the implementation
    - Classically referred to as *economy of mechanism*
  - Category: Prevention

“We’ve seen security bugs in almost everything: operating systems, applications programs, network hardware and software, and security products themselves. This is a direct result of the complexity of these systems. The more complex a system is—the more options it has, the more functionality it has, the more interfaces it has, the more interactions it has—the harder it is to analyze [its security]”. —Bruce Schneier

FS: Use fail-safe defaults

- Some configuration or usage choices affect a system's security
  - The length of cryptographic keys
  - The choice of a password
  - Which inputs are deemed valid

- The default choice should be a secure one
  - Default key length is secure (e.g., 2048-bit RSA keys)
  - No default password: cannot run the system without picking one
  - Whitelist valid objects, rather than blacklist invalid ones
    - E.g., don't render images from unknown sources

Blog: SANS Security Trend Line

"... whitelisting on servers and single function servers or appliances has proven to cause near zero business or IT administration disruption"

FS: Do not expect expert users

[Only] Computer scientists and drug dealers have users — R. David Lankes

- Software designers should consider how the mindset and abilities of (the least sophisticated of) a system's users will affect security

- Favor simple user interfaces
  - Natural or obvious choice is the secure choice
  - Or avoid choices at all, if possible, when it comes to security
  - Don't have users make frequent security decisions
  - Want to avoid user fatigue
  - Help users explore ramifications of choices
    - E.g., allow admin to explore user view of set access control policy

The New York Times

Hackers Breach Security of HealthCare.gov

By ROBERT PEAR and NICOLE PERLROTH  SEPT. 4, 2014

WASHINGTON — Hackers breached security at the website of the government’s health insurance marketplace, HealthCare.gov, but did not steal any personal information on consumers, Obama administration officials said Thursday. …

Mr. Allbright said the hacking was made possible by several security weaknesses. The test server should not have been connected to the Internet, he said, and it came from the manufacturer with a default password that had not been changed.
Passwords

- **Goal:** easy to remember but hard to guess
  - Turns out to be **wrong** in many cases!
    - Hard to guess = Hard to remember!
  - Compounding problem: **repeated password use**

- **Password cracking tools** train on released data to quickly guess common passwords
  - Project Rainbow, [http://project-rainbowcrack.com/](http://project-rainbowcrack.com/)
  - many more …

- **Top 10 worst passwords of 2013:**
  1. 123456
  2. password
  3. 12345678
  4. qwerty
  5. abc123
  6. 123456789
  7. 111111
  8. 1234567
  9. iloveyou
  10. adobe123
  [from SplashData]

Password Manager

- A password manager (PM) stores a **database of passwords**, indexed by site
  - Encrypted by a **single, master password** chosen (and remembered) by the user, used as a key
  - **PM generates complicated per-site passwords**
    - Hard to guess, hard to remember, but the latter doesn’t matter!

- **Benefits**
  - Only a single password for user to remember
  - User’s password at any given site is hard to guess
  - Compromise of password at one site does not permit immediate compromise at other sites

- **But**
  - Must still **protect and remember strong master password**

Password Strength Meter

- **Gives user feedback on the strength of the password**
  - Intended to measure guessability
  - Research shows that these can **work**, but the design must be **stringent** (e.g., forcing unusual characters)

Better together

- **Password manager**
  - **One security decision**, not many

- **Password meter**
  - Users can **explore ramifications of various choices** by visualizing quality and reasoning of password
  - **Do not permit poor choices** (or reduce the chances of them) by enforcing a minimum score
Phishing

• **User is tricked** into thinking that a site or e-mail is legitimate, rather than a scam
• And is then tricked into installing malware or performing other harmful actions

---

Trust with Reluctance (TwR)

• **Whole system security** depends on the secure operation of its parts
  • These parts are trusted

• So: **Improve security by reducing the need trust**
  • By using a better design
  • By using a better implementation process
  • By not making unnecessary assumptions
    - If you use third party code, how do you know what it does?
    - If you are not a crypto expert, why do you think you can design/implement your own crypto algorithm?

• **Categories:** Prevention and mitigation

---

Phishing

• **Failure:** Site or e-mail not (really) authenticated
  • Internet e-mail and web protocols not originally designed for remote authentication
  • Solution is hard to deploy
    - Use hard-to-fake notions of identity, like public key cryptography.
    - But which system? How to upgrade gradually?

---

Design Category:
Trust with Reluctance

---

Phishing

---

TrustedBank™

Dear valued customer of TrustedBank,

We have received notice that you have recently attempted to withdraw the following amount from your checking account while in another country: $150.25.

If this information is not correct, someone unknown may have access to your account. As a safety measure, please visit our website (www.trustedbank.com) to verify your personal information:

http://www.trustedbank.com/trustedbank/authenticate

Once this has been done, our fraud department will verify the accuracy of your account. We are happy you have chosen us to do business with.

Thank you,
TrustedBank
**TwR: Small TCB**

- Keep the **TCB small** (and simple) to reduce overall susceptibility to compromise
  - The trusted computing base (TCB) comprises the system components that **must** work correctly to ensure security
  - **Category:** Prevention

- **Example:** Operating system kernels
  - Kernels enforce security policies, but are often millions of lines of code
    - Compromise in a device driver compromises security overall
  - Better: Minimize size of kernel to reduce trusted components
    - Device drivers moved outside of kernel in micro-kernel designs

---

**TwR: Least Privilege**

- Don’t give a part of the system more privileges than it needs to do its job ("need to know")
  - **Category:** Mitigation

- **Example:** Attenuate delegations
  - Mail program delegates to editor for authoring mails
    - vi, emacs
  - But many editors permit escaping to a command shell to run arbitrary programs: too much privilege!
  - Better Design: Use a restricted editor (pico)

---

**Failure: Large TCB**

- **Security software** is part of the TCB
  - But as it grows in size and complexity, it becomes vulnerable itself, and can be bypassed

**Lesson:** Trust is Transitive

- **If you trust something, you trust what it trusts**
  - This trust can be misplaced

- **Previous e-mail client example**
  - Mailer delegates to an arbitrary editor
  - The editor permits running arbitrary code
  - Hence the mailer permits running arbitrary code
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.

Rule: Input validation

- Input validation is a kind of least privilege
  - You are trusting 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)

Libraries

- One never writes an entire application on its own
- We delegate to libraries

Dangers:
- The library will do the wrong thing with certain inputs
- The library will delegate to code that does the wrong thing
- Need to justify trust

Validating library inputs

- Philosophy of C library: I trust my inputs
  - `fwrite(fp,"foo")` — the `fwrite` code assumes that `fp` will be a valid file pointer
    - Not closed, not NULL, etc.
  - `strcpy(dst,src)` — the `strcpy` code assumes that `dst` has enough space to contain `src`'s contents, and that `src` is null-terminated
    - Etc.
- Many libraries trust their inputs
- So it's the client's responsibility to check them
Library validating its input

- Library routines can also check their own input
- This is what our `catwrapper.rb` program was doing — didn’t expect the web server to do it
- Likewise `fwrite` could have been written to check that its `fp` argument is valid (e.g., non-NULL).

- In general: validation in the library is safer
  - Programmers forget, and are lazy
  - One library, many clients: Fewer chances for mistakes

- But validation may be context dependent
  - and is less efficient when client already knows it’s valid

TwR: Promote Privacy

- A good overall system goal is to restrict flow of sensitive data as much as possible
- Doing so promotes privacy by reducing trust/privilege
- **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.

TwR: Compartmentalization

- Isolate a system component in a compartment, or sandbox, reducing its privilege by making certain interactions impossible
- **Category:** Prevention and Mitigation

- **Example:** Disconnect student records database from the Internet
  - Grant access only be direct terminals

- **Example:** Seccomp system call in Linux
  - Enables compartments for untrusted code

**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?
Minimize privilege

Example:

- Our `catwrapper.rb` program was able to read, write, and delete all files.
- We could reduce the permissions
  - on the directory to forbid deletions, and
  - on the file to forbid writing
- Puts less trust in `catwrapper.rb` to do the right thing — *defense in depth*

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: Can only use already-open file descriptors
  - *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
    - Policy akin to Berkeley Packet Filters (BPF)
  - 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

Design Categories: **Defense in Depth** and **Monitoring/Traceability**
Defense in Depth (DiD)

- **Security by diversity**
  - If one layer is broken, there is another of a materially different character that needs to be bypassed
  
- **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 for avoiding low-level vulnerabilities

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 Newsbite for latest top threats
  - Academic and industry conferences and journals for longer term trends, technology, and risks

Failure: Authentication Bypass

- **(Poor) passwords can be guessed**
  - bypassing authentication process intent

- **Passwords can be stolen**
  - Defense in depth: Should encrypt the password database
    - Assumess that compromise is possible, and thus requires additional defense

Failure: Broken Crypto Impl.

**Getting crypto right is hard**

Remote Timing Attacks are Practical

- Dan Boneh
  - dabo@cs.stanford.edu
- David Brumley
  - dbrumley@ca.cloud.net

**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 demonstrate that timing attacks can be used to extract private keys from 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.

**Timing channel**

Use vetted implementations and algorithms

**Widespread Weak Keys in Network Devices**

Both the attacker and the server run in different domains. With that context, we demonstrate that common SSL applications such as a web server (Apache+SSL) and SSL tunneling (SSH) make use of the same random number generator. This allows an attacker to set up a timing channel within the same domain, and compromise the server by running the attacker's code on the same machine. A timing channel that breaks two domains on the same machine might give management access to the domain of each domain. Two vulnerabilities are used to actively exploit the attack to extract the secret key belonging to the domain.

**Poor randomness**

USENIX Security’12
Monitoring and Traceability

- If you are attacked, how will you know it?
  - Once you learn, how will you discern the cause?

- Software must be designed to log relevant operational information
  - 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

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

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)
{
    /* 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;
}
```

replace uses of char* with struct mystr*
and uses of strcpy with str_copy

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
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

Attacks?

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
Other VSFTPD notables

- **Secure sockets** option, for encrypted connections
  - But **not turned on by default**: “OpenSSL is a massive quantity of code which is essentially parsing complex protocol under the full control of remote malicious clients. SSL / TLS is disabled by default, both at compile time and run time. This forces packagers and administrators to make the decision that they trust the OpenSSL library. I personally haven’t yet formed an opinion on whether I consider the OpenSSL code trustworthy.”
- **Eschews trusting other executables**
  - Doesn’t use `/bin/ls` for directory listings