UMBC

TRAINING CENTERS

Secure Programming

TCPRG4005-2021-05-20

Table of Contents

Chapter 1. Introduction

If you have written software, someone will try to exploit the bugs you have written. If you have tested software, someone will try to exploit the bugs you missed. No one who has written software can claim to write code without errors. No one who tests software can claim to find all errors.

IMPORTANT All software has bugs and exploits.

All security is attempting to raise the floor of difficulty such that exploits are impractical to find or subvert. This may take the form of 4096-bit encryption. It may take the form of a locked data center. But, unfortunately, it only takes one hole to sink a ship.

> ...Remember, we only have to be lucky once. You have to be lucky always.
>
> — IRA to Margaret Thatcher, 13 October 1984

Security comes not just from "programming well". A different kind of mindset and strategy is required to built robust, secure software. A large part of that mindset is accepting that bugs will be written.

But, by using strategies and techniques designed for safety and security, many such security defects can be prevented in the first place. Much software is written by amateurs. The modern Internet was mostly written and implemented by college students. This is not a bad thing! The open and egalitarian nature of software development is beneficial to all.

But this low floor to programming means that there are many considerations, especially security-related, that a new developer may just not be exposed to. This is not the fault of the amateur, nor the fault of the senior programmer. Like all programmers, they will write bugs. By accepting that fact, they can begin to mitigate those bugs.

Security From the Ground Up

One of the first ideas to internalize when writing secure code is that security is not an add-on feature. Security holes can creep in at any point in the Software Development LifeCycle (SDLC).

SDLC

1. Gather Requirements
2. Draft Design
3. Implement Features
4. Test for Defects
5. Deploy to Production
6. Maintain Services

For instance, a client may have a requirement that is intrinsically insecure, like “The CEO needs to be able to review the credit card numbers of all customers at any time.”. Careful discussion, review, and design can eliminate most of these.

These can be further mitigated through the presence of policy and strategy documents. It is forward-thinking to have a policy on how Personally Identifiable Information (PII) is handled; how financial information is stored or retrieved; or how authentication must be performed.

Security is not a patch, nor a plugin, nor an add-on. Secure software takes effort and diligence. Secure practices, algorithms, and operations must be a part of the software system from the first day.

Exercises

Exercise 1

Reflect on how many bugs you have inadvertently written in your career. Accept that you will write buggy code in the future. What techniques can you use to find these bugs? How can these be enhanced or extended?

Exercise 2

What is a computer program? Is it the description? The bytes of source code? The instructions? Its execution?

Exercise 3

Should software vendors be shielded from product liability? Why or why not?

Exercise 4

For each phase of the SDLC, describe two security bugs that could be introduced.

Exercise 5

How separate is your development department from operations? How can you politically influence your organization to consider security as a convern overarching both?

Exercise 6

A business requirement forces the deployment of a closed-source, third party application that makes use of highly unsecure network protocols. How do you deploy the application to mitigate as much risk as possible?

Chapter 2. Cryptography

Knowing a bit of the history and implementation of cryptography will aid in using it. Knowing the weaknesses of cryptography are even more important, especially how attacks against them are used. But while this section deals with cryptosystems, there is on major point:

DANGER: **Never roll your own cryptographic implementations.**

Cryptosystems and algorithms look interesting and fun. Many are simple enough that they could be coded in an afternoon. But any implementation is subject to bugs, both in output and (worryingly) in cryptographic weakness. Unless the developer is both experienced in the language in question, and fluent in cryptography, such a programming attempt is almost certainly going to result in weakness in the implementation.

Part of being humble about one's skill and ability is knowing when bugs would be unacceptable, and taking steps to mitigate against them.

Caesar Cipher

The **Caesar Cipher** is the oldest known cipher. It shifts the plaintext alphabet by certain number of places, wrapping around to the beginning if needed.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J

ATTACKATDAWN → KDDKMUKDNKGX

The **key** of a given instance of the Caesar Cipher is the shift of the plaintext alphabet. Variations on this exist, such as reversing the alphabet as well as shifting it.

The Caesar Cipher is quite vulnerable. To begin with, there are only 25 possible cipher alphabets (51 with reversal). This is well within the realm of brute force. When a given word is known, the rest of the cipher alphabet can easily be derived. Further, it is just a more restricted version of the more general Substitution Cipher, which is vulnerable to frequency analysis.

Substitution Cipher

The **Substitution Cipher** is a refinement of the simpler Caesar Cipher. Each letter of the plaintext alphabet is mapped to an arbitrary ciphertext alphabet letter. This fixes the limited number of cipher alphabets that weakened the Caesar Cipher.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
C	I	D	J	S	Z	B	O	N	A	M	T	K	W	F	P	H	X	R	E	Q	U	L	Y	V	G

ATTACKATDAWN → CEECDMCEJCLW

The **key** of a given instance of the Substitution Cipher is the entire ciphertext alphabet. This can be simplified by using a smaller “keyword” to start the ciphertext alphabet. Note that the keyword may not have repeated letters.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
S	W	O	R	D	F	I	H	J	K	L	M	N	P	Q	T	U	V	X	Y	Z	A	B	C	E	G

ATTACKATDAWN → SYYSRLSYRSBP

Despite the improvements over the Caesar Cipher, the Substitution Cipher is also very weak. It is cracked easily by **frequency analysis**, because different letters in plaintext are used at different frequencies.

Vigenére Cipher

The **Vigenére Cipher** is a series of interleaved Caesar Ciphers. Every ***N***th letter in the plaintext uses the same Caesar Cipher, where ***N*** is the length of the key.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
1	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
2	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
3	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
4	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
5	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
6	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
7	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
8	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
9	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G

ATTACKATDAWN → SPHRFPILKSSB

The **key** is a keyword that determines the Caesar Cipher for each column of text. The keyword is repeated over and over again for the length of the plaintext.

The Vigenére Cipher, also known as “Le Chiffre Indéchiffrable” (“The undecipherable cipher”) was eventually broken by Charles Babbage. By trying different possible key lengths, the individual Caesar Ciphers fall to frequency analysis.

RSA Public-Key Cryptography

In previous systems, both decryption and encryption were dependent upon the secret key. This is known as a **symmetric key**. The same key is used for encoding a message as decoding a message. It was not possible to send a secret message without knowing the key.

In 1977, the RSA public-key algorithm was defined, which uses **assymetric keys**: a public key which is used to encode messages, and a different private key which is used to decode messages. This means that the encoding key can be known by anyone. As long as the decoding key remains private, highly secure communication is possible.

RSA involves quite a bit more math than previous ciphers; it uses exponentiation and modular arithmetic as a kind of "trap-door function", a function that is easy to calculate, but hard to reverse. A metaphor would be a parking lot full of cars. For a given spot, it's easy to walk up to the car, open the door, and write down the serial number. But, given a serial number, there is no fast way to determine which car in the lot corresponds to it.

RSA Process

1. Pick two prime numbers, p and q.
2. Compute their product, $pq = n$. n is public.
3. Compute the least common multiple of p - 1 and q - 1, $\lambda(n)$. This is straightforward to do when p and q are known, but difficult if they are not.
4. Pick a number e that is relatively prime to $\lambda(n)$. e is public.
5. Compute what number d yields $de \bmod \lambda(n) = 1$

To encode a message, each block of the message receives random padding. Then, each block m is run through the calculation of $c = m^e \bmod n$. c is the ciphertext for that block.

In this example, each block is one letter (a **very** poor choice). The random padding is a multiple of 26 added to each letter, with (n, e) = (221, 35). These values for (n, e) are comically low and only for demonstration purposes.

ATTACKATDAWN → 144, 28, 132, 14, 191, 97, 14, 145, 179, 79, 186, 170

```
cipher = [(ord(c)-64 + 26*randrange(1,8))**35 % 221 for c in 'ATTACKATDAWN']
```

The decoder needs to run the calculation of $m = c^d \bmod n$. Note that d was not public. Once any padding is removed, the original block is revealed. In this case, d was 11.

144, 28, 132, 14, 191, 97, 14, 145, 179, 79, 186, 170 → ATTACKATDAWN

```
letters = [chr(64 + (c**11 % 221)%26) for c in cipher]
```

While there are some slight areas of leverage when it comes to cracking RSA, the base difficult problem (factoring out large primes) puts the majority of the algorithm outside the realm of easy exploitation. Most attacks involve side-channels such as timing, or a poor choice of exponent **e**.

Diffie-Hellman Key Exchange

All technologies thus far rely on a secret key. To perform decryption, the secret key is needed. But, what if all channels of communication are monitored? The **Diffie-Hellman key exchange** allows two parties to establish a shared secret key. It does this even though communication between the two may be monitored (but not controlled; it is vulnerable to man-in-the-middle attacks).

Diffie-Hellman Process

1. Pick a prime number p, and a generator g for the group Z_n. Both p and g are public.
2. Each participant picks a secret number: call one a and the other secret b.
3. The one with secret number a computes $A = g^a \bmod p$ and transmits A to the other party. A is public.
4. The one with secret number b computes $B = g^b \bmod p$ and transmits B to the other party. B is public.
5. The one with secret number a computes $s = B^a \bmod p$ as the shared secret.
6. The one with secret number b computes $s = A^b \bmod p$ as the shared secret.

The key exchange (really key synchronization) does not define a cipher. It is merely a way of synchronizing a secret over a monitored channel.

As an example with laughably small numbers, let (p, g) = (257, 12). Person A picks a secret number a and calculates A, yielding A = 29. Person B picks a secret number b and calculates B, yielding B = 190. After transmitting those values publicly, they now calculate the shared secret. Person A calculates s as $190^a \bmod 257 = 242$. Person B calculates s as $29^b \bmod 257 = 242$. The two parties have agreed on a shared secret number.

Exercises

Note that these exercises are not about implementing the encoding portion of a cipher. They are concerned with decoding or cracking only. **Never roll your own cryptographic implementations.**

Exercise 1

Write a program that prints all 26 Caesar Cipher rotations of a given ciphertext.

Exercise 2

Write a program that does English letter-frequency analysis of a Substitution Cipher ciphertext and prints out the most likely key.

Exercise 3

Write a program that does letter-frequency analysis to determine which language (of English, French, Spanish, or German) a given Substitution Cipher ciphertext may be.

Exercise 4

For RSA, suppose the public values (***n***, ***e***) are known to be (28108357, 73). Determine the value ***d***.

Exercise 5

For the Diffie-Hellman key exchange example, determine the secret numbers ***a*** and ***b***. Is there any faster way other than checking all different possible secret numbers?

Chapter 3. Authentication

Authentication is determining that the user attemping access is a valid user.

Initial authentication for secure systems often involves a physical component: perhaps an employee must be in the office to receive their credentials or tools.

Authentication can be subverted through weak passwords, data breaches, or programmatic gaps in the authentication usage.

Weak Passwords

Passwords have long been a easy exploit route. For decades, password requirements have been trending towards complexity. “Must contain a digit, punctuation, lower- and uppercase letters, and no common words.” Unfortunately, this added complexity for humans often leads to circumvention of the intent.

On a standard keyboard, there are 26 uppercase letters, 26 lowercase, 10 digits, and 32 punctuation symbols. That means each keystroke has a choice of 94 glyphs. That means there exist 6,095,689,385,410,816 eight-character passwords.

The best way to add memorable complexity to passwords is by increasing their length. Most English speakers have a vocabulary of 20,000-35,000 words. That means there exist 160,000,000,000,000,000 four-word passwords, a increase of a few orders of magnitude over the strong eight-character passwords.

Restrictive Passwords

Password restrictions will limit the space an attacker needs to search. For instance, many older systems (especially banks) may forbid the use of punctuation. Or may treat upper- and lowercase letters as the same. That would drop the number of eight-character passwords down to only 2,821,109,907,456.

More worrisome that the smaller search space is what forbidding characters implies: that the system inquestion is **sensitive** to storing passwords with such forbidden characters. In other words, it is highly likely that these passwords are stored unencrypted. Even worse, it may be that some layers of the overall system are subject to SQL Injection attacks.

Offensive Security

Investing some regular time with a password cracker is actually very useful from a security perspective. Run all passwords in the system through a password cracker at regular intervals. Any passwords that get cracked should require the account password be reset, because the password in question was indeed crackable. By proactively checking for weak passwords, any password exploit becomes less likely, and any data leak is harder to exploit. Even better would be to run each reset of a password through such a cracker, if performance permits.

Data Breaches

At some point, critical data (such as password hashes) will be leaked. This immediately puts the organization into damage mitigation. A prepared organization will already have a plan or strategy in place as to what to do when this happens.

To begin with, all passwords **must never** be stored in plaintext. Nor should passwords be stored in any way that they are "recoverable".

IMPORTANT	Passwords must **never** be reversible nor recoverable from what is stored.

If a leak of plaintext or reversible passwords occurs, any attacker has unfettered access to accounts. And, to make matters worse for the users of those accounts, many users will use the same password in multiple services. So, a leak of `tiptoppoker.com` accounts can reveal emails/passwords for `bestbank.com`. Passwords must never be stored in a way that they can be recovered.

Some services may provide some form of a recovery phrase that is stored in plaintext This may allow a customer service representative to help remind a valid user of their recovery phrase. Any such process must have very clear policies in place as to how such a reminder may be given.

Hashing

Passwords should be encrypted using a one-way hash function. A good one-way hash function transforms arbitrary input to a fixed-size output. It should have the following properties:

1. A given key should always produce the same hash.
2. Given a hash, it is practically impossible to produce a key that would generate it.
3. **Collisions** between different keys are rare, and practically impossible to create intentionally.
4. Changing a single part of the input changes an unpredictable majority of the output.

For instance, using the SHA-1 hash function produces the following:

```
sha1('abb') -> 0xc64d3fcde20c5cd03142171e5ac47a87aa3c8ace
sha1('abc') -> 0xa9993e364706816aba3e25717850c26c9cd0d89d
sha1('abb') -> 0xc64d3fcde20c5cd03142171e5ac47a87aa3c8ace
```

Reversing the key from the hash is extremely difficult, and a single-bit change of input corresponded to a large change in the output. Not pictured is the fact that SHA-1 is now known to be vulnerable to collisions.

So, rather than leaking a list of plaintext passwords, the hashes of those passwords are leaked. Since the hash is (theoretically) practically impossible to reverse, the actual passwords are safer. But not yet safe.

Salts

With only a hash function, there is still vulnerability: hashes of short or common passwords can be generated ahead of time. Then, when encountering the SHA-1 hash of `5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8`, such a lookup can reveal that the original key was `password`. And, if two users share the same password, that will be obvious when the two hashes have the same value.

So, to address this, each key should be padded with random bits known as a **salt**. This will prevent any precomputation of common or short passwords. Additionally, if two users share the same password, but have different salts, their hashes will be different. The salt must be stored with the hash, but should be random for each user and the user does not need to be aware of the salt or its value.

```
sha1('password' + '9kpnw43x') -> 0xa43d18b09df04646743795b63aae24ce20e039b9
# Store '9kpnw43x', 0xa43d18b09df04646743795b63aae24ce20e039b9
sha1('password' + '1vdl4ef8') -> 0x009aff16014e3047a89775e2a89c9637c0254745
# Store '1vdl4ef8', 0x009aff16014e3047a89775e2a89c9637c0254745
```

Even if the salt is leaked along with the hash, any attacker must generate a new set of brute-force passwords with the salt. Precomputing passwords is no longer an efficient means of attack. But, bruteforcing, especially with common passwords, is still a good route of attack.

Notification

While embarassing, user notification is vitally important. Once a user's password hash is leaked, the user must be informed, and their account marked as needing re-authentication. Forcing the user to choose a new password upon login is vital. A message reminding them to change the password if it used elsewhere is also important.

Depending on the capabilities of a user account, it may even be prudent to lock that account until some form of secondary authentication is available.

Authentication Misfires

There are a number of ways that a given program might misfire with regards to authentication. Some of these are obvious, but many less so.

Fast Authentication

Usually, computer performance should be as fast as possible. But authentication should specifically be slow enough to prevent automated attacks.

If a system can reject an invalid key in 0.000001s, that means one million attempts can be made every second. On the other hand, if it takes 0.0001s or even 1s, that greatly increases the time that a brute-force attack may take, generally to the point of not being worthwhile.

Some systems address this with increasing rejection times: Perhaps the first rejection takes 1s, then the second 2s, then 3s, 5s, etc. This may cause some slight inconvenience to the valid user, but makes brute force attacks all but impractical. As a downside, an attacker could use this as a denial-of-service attack, albeit perhaps just on a single account.

Modern hash functions can accept a "work factor", which forces them to do more work for a given key. This is useful in both preventing automated attacks as well as preventing speedy decryption in case of a password hash leak. While many texts may label a fast hash function as ***desirable***, that is not the case for authentication.

Timing Attacks

Related to fast authentication is **inconsistent rejection time**. A password check could be incorrectly coded thusly:

```
# BAD Practice: Opens key to timing attack
# BAD Practice: Not hashing passwords
if attempt == 'swordfish':
    authenticate()
```

With this simple authentication check, the system is now vulnerable to a timing attack: the attempt `swordfission` will take longer to be rejected than the attempt `letmein`. The way that strings or arrays are compared is done with speed in mind: if the first characters do not match, it is impossible for the whole strings to match, so most implementations will stop processing at that point. By trying different inputs and measuring the time to rejection, the actual password can be discovered. The true mitigation would be to check all positions in the password, regardless of whether the earlier ones were already missed.

```
# STILL BAD Practice: Subtle timing attack
# BAD Practice: Open-by-default
# BAD Practice: Not hashing passwords
success = True

for k, a in zip('swordfish', attempt):
    if k != a:
        success = False

if success:
    authenticate()
```

Trusting User-Supplied Authentication Status

With most server/client architectures, the client will need some form of authentication to progress. Any authentication system must track the client's authentication on the server, rather than relying on the client.

This seems a simple enough requirement, but is surprisingly common in the wild. It may be as obvious as expecting `auth=1` in client-side messages. Or it can be a subtle as accepting an older authentication token left on a shared machine.

Exercises

Exercise 1

Make a list of 15 random, uncommon words. Compare against an online list of the 1000 most common English words. How many were not on the list? What does this say about using uncommon words as a passphrase?

Exercise 2

For your organization, draft an incident plan for when password data are leaked.

Exercise 3

Write an authentication routine which tests a submitted word against a static secret, which is resistant to timing attacks.

Chapter 4. Authorization

Authorization is determining the privilege being executed by a user is granted to that user. When analyzing a system, be cognizant that not ***everything*** they do requires authorization. Carefully enumerating the capabilities of a privileged user will aid in event logging.

Authorization can be subverted through various programmatic gaps, and many social ones. Securing an application against social attacks is slightly out of the scope of this course. The largest of the programmatic gaps involve trusting the user.

Trusting User-Supplied Authorizations

It is truly impressive how many web applications can be exploited by adding `?admin=1` to a given URL. This is the equivalent of getting into a movie theater for free by carrying a ladder.

Any privilege system cannot rely on user-provided privileges. These privileges must be gifted by an authorization system. There is no reason for a user-exposed system (such as cookies, clients, etc.) to maintain the privilege list. If the user manages their own privileges, they can alter them.

An ideal authorization system should check the role against its own maintained list of privileges.

From an architectural perspective, the authorization system should be queried for each action or view that requires elevated privileges. When it comes to really engineering-in security, the authorization system could also be aware of the authentication system. In other words, it can keep track of which clients or users are currently active. This helps harden the authorization system in case a user finds a way to modify their roles as part of an attack.

Failing to check Authentication and Authorization

A possible mistake when performing authorization is failing to check authentication! The values of particular roles (e.g., zero) that are high-privilege may also be defaults. With no authentication data, the default values can result in elevated privilege.

```
// BAD Practice: $_SESSION may be unset!
if($_SESSION['uid'] == 0) {
    // Execute admin-only operation
}
```

```
// BAD Practice: $uid is only populated when a user is authenticated
if($_SESSION) {
    $uid = $_SESSION['uid'];
}

if($uid == 0) {
    // Execute admin-only operation
}
```

The best solution is to prevent all unauthenticated access to areas that must check for authorization. The very least should be to set appropriate, invalid values

```
// BETTER Practice: Obviously invalid default value
$uid = -1;
if($_SESSION) {
    $uid = $_SESSION['uid'];
}

if($uid == 0) {
    // Execute admin-only operation
}
```

```
// BETTER Practice: Chain authorization to happen after authentication
if($_SESSION) {
    $uid = $_SESSION['uid'];
    if($uid == 0) {
        // Execute admin-only operation
    }
}
```

```php
// BETTER BETTER Practice: Prevent unauthenticated access
if(!$_SESSION) {
    exit();
}

// RISKY: Needs earlier code context to confirm as safe.
$uid = $_SESSION['uid'];
if($uid == 0) {
    // Execute admin-only operation
}
```

Failing to Contain Actions to Subprograms

When it comes to executing an action that requires elevated system privileges, the authorization system should ideally delegate the work to a subprogram that has those privileges and *only* those privileges. This is done as a hardening measure. It does not by itself guard against a specific vulnerability, but adds layers of defense to the application.

As an example of such a vulnerability, imagine that an attacker had discovered a shell injection attack against the application. A clever attacker could cause a normal system action (say archiving a file) to run some other command. If the application runs as root or administrator, the system is effectively compromised completely. But, suppose that the "archive-a-file" action is a subprocess run by a system account with limited privileges and ownership. In this case, the attacker may have suborned this account, but the damage is limited to only what that account's capabilities are.

This ties into the general architecture trend of making applications interactive, untrusting components. The overall application, once it has authorized an action, delegates the work to a different process running as a different system user.

This can be further hardened, not only with file permissions, but also with resource limits. This would prevent possible denial-of-service attacks, such as filling up the disk or using too much CPU time. Introducing system quotas to the subprograms should be part of any security profile.

Exercises

Exercise 1

Think about a software system you use that involves users with different privileges. Draft a list of activities that should involve authorization.

Exercise 2

Design a web service that allows a user to delete old syslog files from the system the web service runs on. How should authorization be handled? What components are required?

Chapter 5. User Management

How are users and accounts managed? Answering this question will help define the roles relevant to the software system.

Users are an unfortunate fact of any successful system. Despite the fact that users may be lazy, insecure, insubordinate, or oblivious, a given system must support them in their goals. Related, of course, is the need to keep their work safe and secure.

Any system must carefully balance the need to keep the system secure with the need to get work done. If a system leans too far in the direction of security, users will find worse ways to circumvent the security than what the security tried to solve! Security should ideally be transparent to a user of the system.

User Creation and Locking

Any system that manages users needs to be able to create new ones and lock existing ones. User creation is something that should generally not be fully automated, but require multiple roles to confirm as an action. Locking, on the other hand, is an action that should be allowed to happen automatically, in response to certain events. Much easier to unlock someone than try to undo the actions of an attacker.

Note that the phrase “locked” rather than “deleted” is used here. When it comes to auditing and review, locking is almost always better than deleting. Further, a mechanism to delete users entirely is ripe for exploit by an attacker.

As an implementation detail, it is best that the “locked” status is ***not*** a separate field on a user. A more robust approach is to strip the user account of all roles it belongs to—this delegates the work back to the authorization system. It is surprisingly easy for a separate field to fail to be checked in every relevant circumstance.

Make sure that the system will update the roles of any active user automatically! Do not have it only populate roles on login! There are a number of systems where logging out and back in were required to gain newly-granted privileges. That means the reverse is also true: a rogue user whose privileges get removed just needs to avoid logging out.

Identifying Role Actions

A given role will require access to a number of authorizable actions. The simplest may be something like changing their password. More complex ones may be auditing and vetting of the actions of other roles.

It is important to note the difference between a role and a user. A user is an actual person using the software. But a user may belong to several roles. A manager, for example, might belong to the **PTO Approver** role as well as the **Pay Raiser** role.

Workflow diagrams can aid the designer in teasing out what particular roles must accomplish. As is usual, a simpler workflow diagram is usually preferable to a large or complex one.

No Group Accounts

Like many topics in the realm of secure programming, financial shortsightedness can result in major holes.

Consider a software company that must pay a per-seat license for a mission-critical web service. If the company buys a single seat, and has five people use the account, they are opening themselves to some serious security problems.

To start with, the password must be shared across multiple people. The attack surface for someone looking to penetrate this account has gotten vastly larger. It just takes one person with the secret to be suborned by an attacker.

Consider further the idea of malicious intent of an internal bad actor. If all logins are the same credentials, how can the system be audited for individual actions? Even with a password manager service, these problems stand.

For some services, this may be overkill. Stock photography and lunch services may be examples where a group account is not too harmful.

But what about services like payroll? Cloud storage of system backups? Sensitive presentations or documents? Corporate social media accounts?

While this may be good advice, how does this apply to software development? It sounds more relevant to an overall security posture. Part of writing secure software is making it **easy** for users to do the secure thing. Software pricing should not encourage users to weaken their security.

In other words, making individual accounts seem out-of-reach for users is a requirements misstep. Additional accounts should be negligible in price compared to the initial account. This may not be a hole in the service software, but it absolutely propogates poor security for its customers.

Authorization Log

Related to authorization (allowed actions of a user) should be logging and auditing of those actions. As discussed elsewhere with regards to logging, the auditing is a key part. It does no good to have authorization events logged if they are not regularly reviewed.

Suspicious behavior is hard to automatically detect. A user reseting their password two days in a row may not be unusual. But ten times in an hour is definitely cause for concern.

Related is the idea of separation of duties: certain tasks should require multiple roles to complete. This means that a nefarious actor would need to control two different roles to achieve their goal. An added benefit is that this hardens the system against internal threats, as well. A user cannot execute sensitive tasks without signoff or follow-up from another user.

Exercises

Exercise 1

For your organization, identify some third-party services. Which of these should require an individual account, instead of shared group access?

Exercise 2

Consider an online shoe store. What kind of roles should exist in such a system? How do the roles interact? What kind of privileges do the roles need?

Chapter 6. Data Validation

There is one key rule when it comes to dealing with user input. This entire course can be summed up with this one rule.

IMPORTANT	Never trust input from the user

Nearly every single security hole comes from trusting, implicitly or explicitly, input from the user. This includes:

- Trusting the user to not enter in too much text
- Trusting the user to not enter certain characters
- Trusting the user to use a given application
- Trusting the user to use a reasonable amount of resources
- Trusting the user to not want to crash the system
- Trusting the user to not want to steal data

Buffer Overflows

For decades, the most common vulnerability in software was the buffer overflow. It is still a risk in any language that allows for unbounded array access, such a C, C++, and Assembly. When using one of these languages, it is important to manually perform boundary checks.

```
// BAD Practice: fails to restrict output to BUF_SZ bytes
char buffer[BUF_SZ];
strcpy(buffer, user_input);
```

There are four major C functions that must be avoided at all costs: `gets`, `sprintf`, `strcpy`, and `strcat`. These will all copy potentially unlimited amounts of memory. The presence of one of these functions (especially the first two) almost guarantee a security vulnerability. Safer versions exist: `fgets`, `snprintf`, `strncpy`, and `strncat`.

```
char buffer[BUF_SZ];
// BAD Practice: fails to restrict output to BUF_SZ bytes
strncpy(buffer, user_input, strlen(user_input));
// BAD Practice: fails to NUL-terminate string in some cases
strncpy(buffer, user_input, sizeof(buffer));
```

It is not enough to just use a safer version of one of the aforementioned C functions. One also has to know how to use them. The safer versions allow for a parameter that limits the amount of memory copied or used. But, as in the above code, it is very easy to get the amount wrong in a way that does not stand out. The size specified must **always** be the size of the destination, **never** the size of the source. This goes for any programming language, not just C. Limit the size of a copy to the capacity of the destination, not the amount of the source.

Another consideration when it comes to C is that all strings must end with a single NUL byte, or 0 value. A string that is unterminated is effectively another form of buffer overrun.

```
// Better Practice: allow no more than BUF_SZ-1 bytes (to allow for NUL)
char buffer[BUF_SZ] = {0};
strncpy(buffer, user_input, sizeof(buffer)-1);
```

SQL Injection

The most common exploit today is that of **SQL Injection**. This kind of exploit is introduced by a programmer "stitching together" a query for the SQL language using strings. It usually takes the following form:

```
# BAD Practice: splicing strings destined for another language's execution
# BAD Practice: assuming no errors
def is_admin(user):
    with db.cursor() as cursor:
        cursor.execute(f'''
            SELECT is_admin
            FROM login
            WHERE username = '{user}'
        ''')
    return cursor.fetchone()[0]
```

The problem here is that the programmer is trusting the user input of their username. A malicious user would submit a different form of username:

```
is_admin("'; SELECT true; --")
```

Because the programmer is using only string concatenation, the resultant query looks like this:

```
SELECT is_admin
FROM login
WHERE username = ''; SELECT true; --'
```

Many variations of this bad input exist, such as `"' OR 1=1 OR ''='"`. The malicious actor embeds characters in the input that have special meaning to the second language, usually SQL. This is problematic for any system where two programming languages need to invoke each other with specific user input. SQL is just the most common second language for this sort of exploit. Shell scripting is another common attack vector.

```
# BAD Practice: splicing strings destined for execution
my($month,$year) = split(' ', <>);
system("cal $month $year");
```

Any language that provides the ability to evaluate its code at runtime also exposes this risk.

```
# BAD Practice: splicing strings destined for evaluation
request = input('Enter a string to be split into words: ')
words = eval(f'"{request}".split()')
```

Exercises

Exercise 1

What language is most of your team's programming done in? What is the largest security risk of using that language?

Exercise 2

Find the buffer overflow. What could be passed to exploit it?

```
enum { BUF_SZ = 128 };

char buffer[BUF_SZ] = {0};
strncpy(buffer, user_command, sizeof(buffer)-1);
strncat(buffer, user_argument, sizeof(buffer)-1);
```

Exercise 3

Find the SQL injection. What could be passed to exploit it?

```
user_year = input('Enter a year')
script = f"cal '{user_year}' >output.txt"
sys.execl(script)
```

Chapter 7. Client-Side Security

Client-side security boils down to one major, already made point:

WARNING Never trust the client.

That said, in many cases the user of the software may need a custom client program. This custom client may connect back to a server application. How can the client application be secured against exploitation? It cannot.

Anything that is given to an end-user must be considered as corruptible. A good example for handling client-side corruption would be modern online game design. Such multiplayer games must be exceedingly strict in what they accept from a client program, and pass it through multiple layers of checks and verification.

Even Metadata and Side-Channel Data From the Client Are Suspect

What Operating System is the client running? This question will not have a guaranteed answer. As the client user has total control over their machine, they can massage any response to whatever they want.

While metadata can be queried and used as a starting point, no business decisions should be made based on that information.

Further, even things that normally would be inconceivable to mistrust are suspect. Consider TCP.

With TCP, data generally comes in as a stream of data. But this is just an abstraction over datagrams by providing an ordering to them. A nefarious client can easily cause denial-of-service attacks by dropping key packets and failing to re-transmit them.

Even timing is exploitable. One client can purposefully slow down its network connection. If the server adjusts the application or user experience to account for this slowdown, the devious client can then *turn off* the slowdown, allowing a flurry of commands or requests to suddenly be prioritized due to the "slow" connection. While this is most common in games, this is an issue for any system that takes timing of requests into account.

Client Software Should Ideally Be Stateless

One of the easier ways to secure client software is to make it stateless. Do not track transition states or temporary data on the client side. Rather, track that information as part of a client session on the server side.

The danger in tracking such stateful data on the client side is that the client can modify that data. While this may not be a direct danger, it can very often result in underexercised code paths being used; a common prelude to exploitation.

The client only knows its session ID. This ID should be a very large and sparse value, to prevent attacks against other users. If session IDs are assigned in order, spoofing another user's session is trivial.

It is generally not enough for the server to only track the session ID. Consider some other metadata to validate against as well, such the current TCP connection, or recent timestamp. While an attacker can spoof this metadata as well, this is part of the layering approach of security. A system cannot be bulletproof, but it can make bullets expensive. This approach means that even a leaked session ID is immediately exploitable to the attacker. Note that any such metadata would need to be reset upon authentication; they cannot be persistent across sessions.

Sanitize Client Data

All client-side data should be scanned and sanitized. This should help prevent unusual cases, or interactions between unusual data ranges. Be as specific as possible: Rather than “a one-byte integer”, make the restriction on a field “0-100”. Note that this would be baked into a sanitization module, ***not*** part of the message sent.

This sanitization layer should ideally be part of the serialization/deserialization of messages between services. That way, it cannot be missed or forgotten. If sanitization is a separate function call, then the programmer has to track which pieces of data have been sanitized and which have not. This can quickly become a source of extremely difficult-to-find security holes.

As a rule of thumb, if a message (client or otherwise) fails sanitization, drop the message (if not the connection entirely). A client that is sending bad data is probably about as recoverable as an attacker looking to subvert the system. Dropping the client connection entirely would generally lead the valid user to a restart of the program, which is desired for them. And for an attacker, they are stymied quickly and often as they explore the attack surface. This also prevents denial-of-service attacks on the sanitizer.

Exercises

Exercise 1

Consider a weight tracking system, that reads in some vitals data from the user. How should these data be sanitized?

Exercise 2

Design a computerized tabulation system that reads in data from voting equipment and produces final vote totals. How does the design need to be hardened against the clients (voting machines)? How do irregular connections affect the design?

Chapter 8. Error Handling and Exception Management

Programs have bugs. Programs also encounter exceptional conditions. No software system can execute without running into possible environmental errors, including:

- Out of disk space
- Network disconnect
- Service unavailable
- Printer on fire
- Out of free memory

Note that these are not bugs! These are artifacts of the environment that the program and its computer must run in. This course will call these situations **errors** rather than the usual term of **exceptions**. This is to better distinguish the programming feature known as **exception handling** from exceptions that are **environmental errors**.

Very few programs will never encounter such a roadblock, so all frameworks have some way of recognizing these situations and recovering from them. However, because these are unusual situations, many software programs have bugs when it comes to dealing with these environmental errors.

There are a couple of key parts of managing these environmental errors.

Fail-Closed vs. Fail-Open

IMPORTANT	Most systems should default to fail-closed.

This means that, if a problem occurs, the system should revert to a closed state, rather than an open one.

Consider a bank vault. If the bank vault loses power, should it automatically unlock? Hopefully this example is ridiculous enough that failing-closed can be clearly seen as desirable for most secure systems.

Code should have a default state of "reject" for authentication and authorization. Defaulting to "valid" is dangerous, as there are likely to be unseen paths in the code that may lead to elevated privileges.

```
// BAD Practice: Should have rejection as default
valid = True

if authenticate() == False:
    valid = False
```

```
// BETTER Practice: Has rejection as default
valid = False

if authenticate() == True:
    valid = True
```

```
# BAD Practice: Fail-open
# BAD Practice: Open to timing attack
# BAD Practice: Not hashing passwords

if all(k == a for k, a in zip('swordfish', attempt)):
    authenticate()
```

Exceptions

A number of languages allow for exception-based error handling. The idea being that a given error may not be correctable at the place where it occurs. Instead, the error should be propogated upward through the call stack until a given code context knows how to handle or correct the error. Exceptions behave in this manner, unwinding the call stack until they are handled explicitly, or the program exits.

A language without exception handling would need to manually propogate such a correctable error. And, if the error was not manually propogated, the program might try to continue on with invalid data or state. This can lead to data corruption or security holes.

Exceptions can be checked or unchecked. In a checked-exception system, each function must note the kind of exceptions that it may throw.

```
// Example of checked exception function signature
public void writeToFile(String path, String message) throws java.io.IOException {
    try(var writer = new FileWriter(path, true) {
        writer.write(message);
    }
}
```

For an unchecked-exception system, any possible exceptions have to be discovered through tooling or careful code inspection. Some systems, like C++, only have an indication of "might throw *something*" or "doesn't throw anything".

```
// Example of unchecked exception function signature
void write_to_file(std::string path, std::string message) noexcept(false) {
    std::ofstream writer{path};
    writer << message;
}
```

Exceptions can make code harder to reason about. This is because each instruction and function call can potentially throw an exception. It is no accident that a number of style guides for C++ completely disable exceptions.

That said, exceptions require less attention on the part of the developer. They can be systematically logged and reviewed, unlike non-exception error handling. A well-written system using exceptions can be easily made robust against errors.

There are a number of programming anti-patterns regarding exceptions.

Pokémon Exception Handling

In an effort to avoid dealing with exceptions, a frustrated programmer may catch every possible exception.

```
# BAD Practice: Gotta catch 'em all
try:
    function_that_may_raise_exceptions()
except BaseException as e:
    logging.info(f'Exception {e} was raised')
```

This is almost always the wrong answer to exceptions. It means that the actual handling of the exception has not been written. This is incomplete code.

An exception-handling block should only catch and process what would be appropriate. Other kinds of exceptions should be allowed to bubble up, to either terminate the program or be handled elsewhere.

```
# BETTER Practice: Catch only what can be handled
try:
    function_that_may_raise_exceptions()
except (FileNotFoundError, DirectoryNotFoundError) as e:
    print(f'Unable to create path: {e}')
    reprompt = True
except PermissionDeniedError as e:
    print(f'Forbidden to create path: {e}')
    reprompt = True
except DiskFullError as e:
    print(f'Out of space in path: {e}')
    reprompt = False
```

Swallowing Exceptions

It is easy for an inexperienced developer to accidentally (or misguidedly intentionally) throw away an exception entirely. Rather than correcting the exception, or at the very least logging it, the exception handler is empty.

```
# BAD Practice: Exception swallowed
try:
    function_that_may_raise_exceptions()
except (FileNotFoundError, DirectoryNotFoundError):
    pass
```

This makes it extremely hard to debug the program when exceptions do occur. When buried deep in library code, this can lead to hours of debugging. Further, if the exception indicates a security concern, it is no longer logged nor addressed.

Error: An Error has Occurred

```
# BAD Practice: Dropping error, stacktrace
try:
    function_that_may_raise_exceptions()
except (FileNotFoundError, DirectoryNotFoundError):
    print('Unable to create path')
    reprompt = True
```

Slightly better than swallowing an exception is removing its context. The common way this happens is when the exception object itself is discarded or ususued. Try to make the exception message or data part of any logging or user display.

Eating the Stack

A similar problem to swallowing an exception is eating the stacktrace. This is commonly done when logging is desired, but the error cannot be corrected at the current location.

```
# BAD Practice: Removes context of original exception
try:
    function_that_may_raise_exceptions()
except (FileNotFoundError, DirectoryNotFoundError) as e:
    print(f'Unable to create path: {e}')
    raise e
```

When `raise e` is reached, it raises a **new** exception whose origin is that line, not the original error that caused the exception `e`. If the exception is consistently hit, this is easy to correct and trace the original error. But on Heisenbugs, this is supremely frustrating, as the only evidence in the logs lacks the critical information.

Every exception-based language is capable of re-throwing an exception, preserving its original cause and stacktrace. This usually involves invoking `throw` or `raise` without any argument. The current exception is then reused

```
# BETTER Practice: Rethrows the original exception
try:
    function_that_may_raise_exceptions()
except (FileNotFoundError, DirectoryNotFoundError) as e:
    print(f'Unable to create path: {e}')
    raise
```

Error Handling

Some languages do not support exceptions, or a given style guide may forbid their use. This means that all environmental errors need to be detected manually and corrected.

```
// Example of error handling
int write_to_file(const char *path, const char *message)
{
    FILE *fh = fopen(path, "w");
    if (!fh) {
        perror("Unable to open file");
        return -1;
    }
    int err = fprintf(fh, "%s", message);
    fclose(fh);
    return err;
}
```

This does result in more code at the call site where an environmental error may occur. The result of each operation or function call that may encounter an environmental error needs to be checked and confirmed.

In a more easily-detectable case, the return value of such an operation may be invalid for future operations: A null pointer, `NaN` values, negative values, etc. This can turn an environmental error into a software bug, since an invalid value is now being used. This is at least discoverable and correctable.

```
// BAD Practice: Not checking result of operations that can fail
void write_to_file(const char *path, const char *message)
{
    FILE *fh = fopen(path, "w");
    fprintf(fh, "%s", message);
    fclose(fh);
}
```

But some languages and functions allow their return value to be dropped silently, which can throw away the evidence of such an environmental error. As shown here in the `fprintf()` call, I/O may fail. The only programmatic way to detect this environmental error is by checking the return value manually.

IMPORTANT Remember the ABCs of error checking: **A**lways **B**e **C**hecking for errors.

Error Messages

Related to handling environmental errors is how to notify the user of such conditions. Writing error messages to be consumed by the end user is actually an important security consideration.

To begin with, these environmental errors are likely places for vulnerabilities to live. An attacker is likely to trigger a number of such error conditions in their search for an exploit. And these error messages could potentially leak valuable intelligence to that attacker.

Prevent Exceptions From User Visibility

In an exception-featured language, it is important to prevent exceptions from reaching the user. Partly, this is making sure that all exceptions are handled appropriately. The other reason for this is to ensure security. Exceptions come with a stack trace of where execution is. To an attacker, a stack trace is a valuable source of program architecture information. Not only that, but some languages will also provide the values of variables, leaking possibly confidential information.

The attacker now has direct knowledge of which functions call others, and that it is possible to crash the system. Even if no security vulnerability is leaked, an exception will end the program, which means it can be used as a denial-of-service attack.

Avoid Implementation-Detailed Syntax Messages

Error messages should be informative. But the implementation details should be kept confidential. This is especially noticable across language boundaries, where a separate language must be evaluated. The classic example would be printing out an entire SQL query in an error message if it fails.

This level of detail ***shouldn't*** be exploitable, but may clue in an attacker to valuable information. At worst, it could lead to the attacker finding a SQL injection attack. Even at best, it reveals a lot of information (table names, column names, servers) that the attacker may have merely been able to guess at before. As always, the goal of security is to make cracking it ***expensive***, as ***impossible*** is out of reach.

Implementation details should generally be scrubbed from error messages. For starters, such details are likely to change, and this duplication will be harder to keep sychronized. Secondly, the details are not relevant to users of the system. Thirdly, the details only aid attackers. It is best for an error message to provide feedback to the **user** of the program as to what needs to be fixed.

Avoid Sensitive Information in Messages

A surprising number of error messages presented to end users include confidential information. Accounts, PII, budget numbers, and passwords are all sensitive pieces of information that at some point have shown up in poorly-written error messages to the end user.

It is often tempting to provide such details in an error message, as it makes it easier to troubleshoot the

issue. But, these data should be scrubbed from error messages to prevent intentional leaks (such as screenshots of the error) as well as accidental ones (shoulder-surfing).

Provide Mitigation Steps to a User

An effective error message must be brief, unique, and directive. An overly long or complex error message will be ignored. A generic message (“A problem has occurred”) makes debugging or error mitigation nearly impossible. Assigning a unique error code to each error case is usually a good idea.

The most useful portion of an error message is telling the user what to do to correct the environment. It might be changing permissions on a file, or picking a different one. It might be installing more memory. It might be “please call us and tell us the following error code number.” It might be telling the user to check the cable.

Providing mitigation steps is not only good user interface, it is also good security.

Exercises

Exercise 1

Consider a function that must open a file and check to see if the string `hello` appears in it. What environmental errors could this function encounter? Which can be processed or handled by the function, and which need to be propogated upward?

Exercise 2

Consider an app that connects to a car to play music. What environmental errors could this app encounter?

Chapter 9. Event Logging

At some point, a system will fall prey to an exploit. Before the damage can be mitigated, the scope of the breach and what systems have been affected must be pinpointed. This can be somewhere between difficult and impossible without the presence of effective logs.

Not only can logs help with this extreme situation, well-maintained logs also assist with debugging and auditing. But, logging is not just a matter of throwing messages into a file. Doing so will lead to a haystack of spurious messages. Better to carefully curate what is logged and retained.

Syslog

`syslog` is one of the oldest, yet also most useful, logging systems. A well-programmed application would be hard-pressed to need more logging capability than what is exposed via syslog.

Syslog is a Unix service that can accept messages from various services at particular alert levels.

```
openlog("authenticator", LOG_PID|LOG_PERROR, LOG_AUTH);
...
syslog(LOG_NOTICE, "Login for '%s' failed", username);
```

First, a connection to the `syslogd` daemon is initiated with `openlog()`. The specification should identify the application, the service, and any desired options. A message is sent to the daemon with `syslog()`, with the desired level.

Syslog Levels

1. EMERG
2. ALERT
3. CRIT
4. ERR
5. WARNING
6. NOTICE
7. INFO
8. DEBUG

Syslog levels run from the `EMERG` "system must shutdown" level to as innocuous as `DEBUG` "debugging". Most applications should only emit `ERR` or levels less critical than that.

When processing messages, `syslogd` can be told to ignore messages less urgent than a given severity, on an application-by-application basis. While there is a small amount of overhead involved in a call to `syslog()`, it remains small.

Syslog is common enough that nearly every language has a simple module to use its interface.

```
import syslog

syslog.syslog('Logging Message')
```

Alerting

It is not enough to log an event. One must also respond to it.

Once events are logged,

Best Practices

There are a number of best practices when it comes to logging events from a system. Most of these should be outlined in a Logging Strategy document which is kept up-to-date and reviewed regularly, just like a Testing or Development Strategy document.

Logging Strategy

Like any systematic approach to problem-solving, effective logging requires a strategy. What items should be logged, and at what level, as well as what items should *not* be logged. Especially important is how aspects of security should logged:

1. Authentication
    - login success
    - login failure
    - logout
2. Authorization
    - impersonation
    - irrevocable action

As well as how PII should be masked in logs.

A logging strategy should also spell out how long logs are kept, and how often they are rotated. Different industries may have legal obligations to keep logs for a specified amount of time.

Logging Levels

Most logging utilities or systems expose different levels of severity. Part of the strategy should be to enumerate acceptable logging levels and their appropriate usage. Most systems, for example, do not need the `EMERG` level, and might only emit an `ALERT` message when the service shuts down unexpectedly.

Centralize Logging

Unlike many services, which may strive for more replication and decentralization, logs work best when centralized. This is done in the name of safety. An exploited system is inherently untrustworthy. It may have its own internal monitoring systems, including logs, suborned to an attacker's desire. And log files are one of the first places an attacker will look to hide their footsteps.

Centralizing logging into a single append-only system means that another system must be compromised for an attacker to modify logs. While this may not be much, it is definitely better than nothing. Some filesystems will even support append-only modes, which is perfect for this sort of system.

Such an independent system should ideally do logging and only logging. This makes its attack surface smaller, by providing fewer services.

Remember That Logs Are Incomplete

Just as important as the events that are logged to a system are the events that are not logged. A developer can forget to log an important event as easily as they forget to handle an exception.

Part of testing a software's function must also involve testing the events it logs. This is particularly critical for systems involving authentication and authorization.

Review Logs Regularly

The most carefully-considered and composed logs are useless without review. True, a meticulous log can be useful after the fact, to determine how events had gone wrong. But even better would be to catch the event while it is happening. This requires not just log review, but some form of monitoring/alert system.

Log review, alerting, and analysis are tasks that must be considered as part of application deployment and maintenance. Which events require immediate intervention? Which events might even involve halting the service or software?

Note that corrective action cannot be exhaustively listed. While relying on some automated event mitigation can be useful, an automated system cannot cover every event. For some pieces of software, an operational team that can respond to unforeseen series of events is vital.

For instance, failed logins are not uncommon. But 1000 per second, in the middle of the night? Such situational responses require a human agent. Especially when the mitigation might be blocking an incoming IP, or reseting a server, or notifying authorities.

This is not to say that automated abatement should be discarded! There are many events where an automatic response makes perfect sense.

Exercises

Exercise 1

Identify a time when a logging system was missing critical events. What additional work was necessary to recover relevant information?

Exercise 2

Sketch out a logging strategy for an online gambling site. What events should be logged? What levels of severity should be used?

Chapter 10. Architecture and Design Patterns

The overall architecture of a software system is usually ripe for possible exploits. While bugs may be the cause of most vulnerabilities, their patches can usually be straightforward. A security misstep in the design, however, is much harder to patch, and may require a radical change.

Consider the Meltdown vulnerability. The architecture of modern processors has been built around CPU caching, allowing for better overall performance. However, when paired with pipelining and out-of-order instructions, allows for a massive, undetectable data breach. Software patches against Meltdown or Spectre result in a 10-30% slowdown. Redesigning hardware that has endured for over 20 years is neither cheap nor easy.

Software, like architecture, can be more easily described using patterns. These are common interactions or constructions that may not have baked-in language syntax, but are implemented over and over again. Similarly, for security concerns, certain **Security Design Patterns** exist. There are dozens of such patterns, only a few will be explored in this course.

Key Secure Design Principles

A number of principles should be kept in mind when it comes to designing a system. This list is not exhaustive, and may be covered in more depth elsewhere in this course. This list is merely to group some of the major points of secure design together.

Least Privilege

Any task or component should only have the minimal set of privileges to complete the job. The military security rule of “need-to-know” is an example of this principle.

Separation of Duties

Require more than one agent to complete a sensitive task. Also known as compartmentalization or segregation of duties.

Defense-in-Depth

All interfaces should have redundancy in security controls or mechanisms, or a layered defense. This requires an attacker to circumvent multiple mechanisms to gain access. Remember that security is not about preventing attackers entirely, but making it economically or tactically infeasible for them to do so.

Fail Safe

A system should protect confidentiality and integrity even in a failed state. This is sometimes also referred to as fail-secure. Upon failing or terminating, a system should not reveal sensitive information about the failure to potential attackers. Note that the information may not necessarily be the attacker’s end goal, but may just provide additional knowledge for creating a successful attack.

Economy of Mechanism

Keep the design of any system small and simple. There are many examples of complex systems having more holes due to that complexity. The Heartbleed vulnerability is an excellent example.

Complete Mediation

Every access to every object must be authorized. Require access checks to an object or piece of data each time a subject requests access. This decreases the chance of mistakenly giving elevated privileges to a requent. Caching permissions does not boost performance enough to justify secure object access.

Open Design

The Security of a system should be open to review. The common example would be Kerckhoff’s Principle: “A cryptosystem should be secure even if everything about the system, except the key, is public

knowledge". A system that lacks outside review of its design will be less hardened than one under scrutiny.

Least Common Mechanism

Minimize common mechanisms between different users and processes. Avoid having multiple subjects sharing mechanisms to grant access to a resource, as sensitive information can potentially be shared between the subjects via the mechanism. A different mechanism (or instantiation of a mechanism) for each subject or role can provide flexibility of access control among various users and prevent potential security violations that would otherwise occur if only one mechanism was implemented.

Psychological Acceptability

If users are inhibited by security mechanisms they fail to accept, they will bypass or disable those mechanisms. Security mechanisms or procedures should be as invisible to the users as possible and introduce minimal obstruction.

Security Architect Role

One pattern for success in software security is that of having a dedicated Security Architect role. This role would be responsible for considering the security of the system from beginning to end.

Just as an Architect may consider use cases and careful interactions, a Security Architect would consider **mis**use cases and **un**careful interactions. For any sufficiently large system, or one with high security risk, such a role makes a lot of sense.

Such a role definitely needs to coordinate with the System Architect. The eventual system, like any system, will need to involve cooperation and compromise from the various stakeholders, and it is the job of the Security Architect to effectively lobby for security. This is a difficult position to be in; security often takes a back seat to business concerns.

Distrustful Decomposition

This pattern's goal is to move separate pieces of functionality into mutually distrustful modules or programs. This way, if a functional unit is compromised, not only is the compromise isolated, but remaining functional units are distrustful of the compromised unit, by design.

This pattern is useful whenever an application must execute with higher levels of permission. Delegating that work to another application means that the first application can be run at lower privilege levels. This makes the attack surface for privileged applications much smaller, frustrating an attacker.

Further, by making the interdependent applications distrustful of one another, validation and logging of elevated requests comes naturally. Some operating systems, such as Unix, naturally use this sort of pattern through small, simple programs and a `fork`/`exec` model of parallelism.

Clear Sensitive Information

For this pattern, a program is dealing with sensitive information. The resources used by the program may be re-used by the system. Those resources may be temporary files, memory, cache, or even pixels on a screen.

The programmer should identify when sensitive information is no longer needed, and explicitly scrub or overwrite those resources. This may involve overwriting memory or disk with zeroes, flushing cache, or blacking out a section of the screen. In some languages, it may be difficult to confirm that the information is indeed scrubbed, rather than optimized away.

Some architectures may have a pool of resources. These pools must be carefully combed for evidence of such sensitive information. Some of these pools (such as database connections) may have very subtle ways of caching information, such as an uncommitted transaction. Simpler mechanisms are usually easier to verify.

Exercises

Exercise 1

Imagine converting a codebase to a different language. What security risks are involved in the migration?

Exercise 2

Imagine porting a Unix application to Windows. What security risks are involved in the migration? What about in the reverse case?

Exercise 3

In the 1990s, an encryption chip known as “Clipper” was developed and promoted for telecommunications. It used a secret, unpublished block cipher known as Skipjack. Each device had a secret key that was deterimined at manufacture time, and escrowed with the government. The secret key could be used to decrypt communications once a given agency established a warrant to do so. What kind of architectural or design flaws seem to exist?

Chapter 11. Threat Modeling

How does one start to think like an attacker? Adopting an adversarial mindset is a useful skill for any security hardening. As a developer, it is very easy to get tunnel vision over the “happy path”, and not give due consideration to exceptional behavior.

The skills of a tester often pay off more than the skills of a developer, in this regard. Testing involves an adversarial mindset, and a drive to try things in unintended ways. It involves exercising all unhappy paths.

Building a **Threat Model** is a structured way of thinking like an attacker. Doing so will generally help create a list of possible threats or attacks for a system. This list can be used to then address or mitigate risk.

Area of Focus

The first step in threat modeling is deciding where to focus: **assets**, **attackers**, or the **system**. Neither of these is better or worse to focus on, but should be chosen appropriately for the project.

Assets

Focusing on assets means protecting items of value to the organization. This is usually an item that is also of value to the attacker. It might be something the attacker wants to access, control, or destroy. An asset might also be modeled as a stepping stone to either of these items.

Assets

- Things to protect
- Things attackers want
- Stepping stones

To model asset-centric threats, the first step is to make a list of such assets. Give careful consideration to each of the three categories of assets.

Attackers

Modeling attackers as the focus on the types and goals of attackers. This can be an extremely useful for communicating risks to stakeholders. By having a specific attacker model, it can be used to explain who would attack an organization and why.

Focusing on modeling attackers is difficult. Unless the attack surface is truly vast, there is not likely to be a huge variety between different attackers in terms of goals or methods.

On the flip side, humanizing the attacker also increases the risk of a stakeholder saying “no one would ever do that.”.

Systems

Focusing on the system is usually the most robust way to start modeling threats. Carefully sketch out a model of the system, and look for interfaces between modules or components. It may be useful to draw out “trust boundaries”, wherein components accept the results of other compenents within the same boundary.

This focus is often useful even during the design and brainstorming phase. The focus and reflection on how the system works can often produce startling gaps in coverage, simply by discussing the components. Knowing what components implicitly trust other components, or that have surfaces exposed in unusual places, is a key understanding for the security engineer.

Peeling Back Layers

Thinking like an attacker often requires “dropping down a level”. Instead of thinking about “a string of characters”, an attacker may view it as “bytes in memory”, “adjacent bits”, or even “voltage signals on a wire”. By seeing through the metaphor to the underlying implementation, an attacker can behave in ways that the programmer did not intend.

A developer that uses `strcpy` may have the high-level metaphor of “copying a string”. The low-level attacker instead views it as “copying bytes”. Thus, the attacker can shoehorn in an exploit that the developer did not anticipate.

A developer that uses TCP may have the high-level metaphor of “a stream of bytes”. The lower-level attacker can instead view it as “a series of orderable datagrams”. This would allow the attacker to issue a denial-of-service attack unconsidered by the developer.

The same sort of approach can be used for testing any kind of security. Yes, the door in the fence may have a lock on it, but what if someone just climbs over the fence?

STRIDE

The STRIDE profile is one of many ways of threat modeling. It names some specific types of attacks, and encourages the user to consider threats that might use any one of those attacks. Originally designed by Microsoft, the STRIDE methodology is fairly straightforward.

S

Spoofing

T

Tampering

R

Repudiation

I

Information Breach

D

Denial of Service

E

Elevation of Privilege

Spoofing

- What ways can authentication be subverted, circumvented, or broken?
- How can somone pose as another person?
- How can a system pose as another system?
- How can a service pose as another service?

Tampering

- What ways can the data or processes be modified?
- What inputs might be unexpected?
- How can sensitive sections be altered?
- Would processing steps out of order be exploitable?

Repudiation

- What data are ripe targets for deletion or destruction?
- What trails exist to track actions?
- How is unusual behavior logged or monitored?
- How would an attacker hide their steps?

Information Breach

- What data are stored in plaintext?
- What data are useful to an attacker?
- How are sensitive data used?

Denial of Service

- What operations are computationally expensive?
- What storage systems are used?
- What CPU/memory/disk/network limits exist for each system?

Elevation of Privilege

- What privileges do the systems need?
- What privileges does each role need?
- In what ways can the authorization system be subverted or broken?
- Can a privilege be invoked accidentally?

Exercises

Exercise 1

A program is described as “it accepts strings from the user and prints them on the screen in triplicate”. Dropping down a level, what is the program doing? What attacks does this lack of abstraction suggest?

Exercise 2

An online application is a poker-playing site. What threats can be brainstormed against it?

Exercise 3

Think of an earlier project. What threats can be brainstormed against it? Does going through the steps of STRIDE reveal any additional possible threats?