latex input: mmd-article-header Title: Notes on Security Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Security, Matthew Koontz, Inherited, Questions CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan C. Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Glossary

- **Access control** --- defining who is allowed access to the data/system
    - You may need to acquire a "*capability*" first
- **Auditing** --- determining *what you did* during your session
- **Authentication** --- the process of verifying that "you are who you say you are", i.e. assuring the message's origin
- **Authorization** --- the process of verifying that "you are permitted to do what you're trying to do"
- **Buffer overflow** --- while writing data to a buffer, a program overruns the buffer's boundary and overwrites adjacent memory
    - They are the basis of many software vulnerabilities and can be maliciously exploited
        - By e.g. overwriting the return address in a stack frame, meaning that execution will "return" to the attacker's address, which contains his data/code
    - Bounds checking can prevent buffer overflows
- **[(Public Key|Digital)] Certificate** --- an electronic document that uses a **digital signature** to bind a **public key** with ID info; e.g. name, org., address, etc.
    - Used to verify that a public key belongs to an individual
    - If you trust the certificate, you also trust that messages signed by the sender’s private key were sent by that person.
    - Managed with a **public-key infrastructure** scheme
    - Issued by a **certificate authority**, a third-party trusted by both the owner of the certificate, and the party relying upon the certificate to validate the ID of the owner
    - To provide a means of determining the legitimacy of a certificate, the sender’s certificate is signed by someone else, whose certificate is in turn signed by someone else, and so on, forming a chain of trust to a certificate that the recipient inherently trusts, called an anchor certificate.
- **Challenge-Response** Authentication --- authentication involves one party presenting a "challenge" and the other providing a "response"
    - E.g. password authentication: "What is your password?" "It is `abc$123`"
- **Checksum** --- allow detection and repair of small number of changes

    - It's goal is similar to a hash function, but the use-case and therefore algorithms used are different
- **Cryptanalysis** --- the study of breaching cryptographic security systems
    - Mathematical analysis of cryptographic algorithms and exploiting weaknesses in their implementation
    - Given encrypted data, retrieve as much info as possible about the unencrypted data

- **Cryptographic hash function** --- takes an arbitrary block of data and returns a fixed-size bit string (the *cryptographic hash value*)
    - any change to the data will change the hash value in an unpredictable way
    - it is often meant to be computationally-intensive to derive any inputs (infinite input space, finite output space) from a given output

- **Cryptosystem**
    1. Any computer system that involves cryptography (e.g. email)
    2. A suite of algorithms needed to implement a particular form of encryption and decryption, generally of the *asymmetric* variety, e.g.
        1. Key generation
        2. Cipher
            1. Encryption
            2. Decryption

- **Dictionary attack** --- trying a dictionary full of *likely* passwords in an attempt to gain unauthorized access
- **Digest** --- the output of a **cryptographic hash function** such as **SHA-1**
- **[Distributed] Denial-of-Service ([D]DOS) attack** --- attempt to make a machine or network resource unavailable to its intended users
    - One common method of attack involves saturating the target machine with external communications requests, so much so that it cannot respond to legitimate traffic, or responds so slowly as to be rendered essentially unavailable.
        - A simple protection is to cache recently seen IP's and throttle responses to repeat requesters (not sure whether this is a "best practice", but I've seen it used on Github)
    - The goal may be extortion, e.g. "give us money or we will DOS you"

- **Exploit** --- a piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic.
- **Firewall** --- block messages based on origin, content, frequency, etc.
    - e.g. drop all packets to or from particular hosts or ports
    - can be done in software, in hardware, on the router, on the host, or in a dedicated unit

- **Hamming weight** --- the number of symbols in a string that are different from the "zero" of the alphabet used
  - Equivalent to the **Hamming distance** from the all-zero string of the same length
  - A.k.a. the l_1-norm of a bit vector
- **Handshake** --- automated negotiation process that dynamically sets parameters of a communications chanel before communication begins
  - After establishing connection, before transferring desired information
  - Agreement on: (e.g.) transfer rate, coding alphabet, protocol, etc.
  - Might just say "got the message" over and over, or might say "data was corrupted, please resend"
  - E.g. TLS has a handshake to determine the symmetric key, which takes place after the TCP handshake completes
- **HMAC** --- *Keyed-Hash Message Authentication Code* --- a **MAC** that uses a *keyed hash function*, i.e. a **cryptographic hash function** parameterized by a secret key
  - This allows the verification of *data integrity* (via MAC) and *authentication* (via hash, which is keyed by a *secret* key)
  - Any normal MAC function (e.g. MD5) can be used, then the secret key parameterizes it
  - Brute force can be used to recover the secret key, but this can be protected against by using a longer key (this slows it down though)
  - If the receiver (improperly) does not hide the time required to calculate the expected HMAC of the received value, a *timing attack* is possible to recover the HMAC, and then forge signed messages
- **HTTPS** --- layers HTTP on top of **SSL/TLS**, so instead of writing bare ASCII to the network, outgoing traffic is encrypted, and then decrypted by the receiving host
  - This is (hopefully!!) how you connect to your bank. You can (and should always) verify this by checking that before you log in, the URL has the little green lock with the name of your bank on it.
- **Initialization vector** --- a pseudorandom number added to a message before encrypting to make this instance of the encryption unique
  - This is useful if you're sending a lot of messages. Consider, if everytime you send a particular message it hashes to the same value, an interceptor with only access to your encrypted messages can simply look at the *pattern* of encrypted values and determine what you're doing.
- **Integrity** --- detect accidental and intentional message changes
- **Keychain** --- say your app needs to access something it shouldn't always have access to. It asks your keychain, via its unique identifier, to open things instead. Your keychain makes a determination that this usage is safe, and performs

whatever action was requested. In this way your app never gains access to the key.

- **MAC** --- *Message Authentication Code* --- a short piece of information used to provide **integrity** and **authenticity** assurances on a message.
    - Often created using a **keyed (cryptographic) hash function** (a.k.a. "MAC algorithm") that a **secret key** and a message, and outputs the MAC
    - The receiver will run the key, message, and MAC through a verification algorithm to ensure integrity and authenticity

- **Modular arithmetic** --- $a \equiv b$ (mod n) **if** $((a-b),)$
    - I guess I might summarize this operation as saying: "a and b are the same distance from being multiples of n"
    - E.g. , $38 = 14, (\mathrm{mod}, 12)$ because $(14-2), == (38-2),$

- **Nonce** --- arbitrary (generally pseudorandom) number used only once in a cryptographic communication
    - E.g. used to **authenticate** users in a way that can't be reused in **replay attacks**
        - First you ask the server for a nonce, then you use the result to encrypt your **password** and send it

- **Private-key / Symmetric Cryptography** --- uses one key for both encryption and decryption
- **Privilege escalation** --- gain elevated access to resources that are normally protected from an application or user
- **Public-key / Asymmetric Cryptography** --- requires a *secret* **private key** and an *openly available* **public key**
    - **Public key**
        - *Encrypt plaintext*
        - *Verify* a **digital signature**

    - **Private key**
        - *Decrypt ciphertext*
        - *Create* the **digital signature**

    - Computationally infeasible to determine private key from public key
        - So public key can be published, and only those with private key may read the messages

- **Rainbow table attack** --- a *dictionary attack* using a precomputed table of reversed cryptographic hash functions
    - using a **salt** makes this attack unfeasible

- **Replay attack** --- someone catches your request to send money to someone else, and they just keep sending that request over and over
- **Salt** --- random data that is used as an additional input to a one-way function that hashes a password or passphrase
    - Protects against *dictionary* and *rainbow table* attacks

- **Secret Key** --- a piece of information that serves as a parameter to a cryptographic algorithm
- **Semantically secure** --- when an attacker cannot distinguish two encryptions from each other even if the they chose the corresponding plaintexts
- **Signature** --- scheme for checking **authentication** and **integrity**
- **Spoofing attack** --- one person or program successfully masquerades as another by falsifying data
- **SSL (Secure Sockets Layer) / TLS (Transport Layer Security)** --- cryptographic protocols designed to provide communication security over the Internet
    - **SSL provides an encrypted TCP connection**
    - **data integrity**
    - **endpoint authentication**
    - First they *assure the counterparty*, aka *handshake*, then they exchange a *symmetric key* which is then used to encrypt data sent between them
    - Often uses port 443 for **HTTPS**
    - *Uses:* eCommerce and banking
    - For more, see `Networking and Network Programming Notes.md`
- **Vulnerability** --- a *security-relevant* **software defect** that can be *exploited*

Notes on Specific Things

OAuth

8/4/15

From Wikipedia

OAuth is an *open standard* for *authorization*. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. It is designed to be used over HTTP. It provides *access tokens* to third-party clients via an authorization server, with the resource owner's approval. This access token is then used by the client to access the protected resources in the resource server. This is the system used when you e.g. log into some third-party website using your Google account.

OAuth 2.0 is the most recent version. Its different implementations are not necessarily interoperable. Its implementations have had numerous security flaws exposed, and the protocol itself has been described by security experts as inherently insecure.

E.g. if a user wants to access their account on an app, the app is going to ask them for a key to their data. The user gets that key from Google and gives it to the app, who then lets them in.

Buffer Overflow (C[++] only, pretty much)

12/8/14

This example is from the Coursera course "Software Security" video "Buffer Overflow"

```
void func(char *arg1) {
    int authenticated = 0;
    char buffer[4]; // quite small
    strcpy(buffer, str);
    if (authenticated) {
        /* do secret stuff */
    }
}

int main() {
    char *mystr = "AuthMe!";
    func(mystr);
}
```

What happens is `mystr` is bigger than `buffer`, so it overwrites `authenticated` to be non-zero, meaning that `if (authenticated)` is `true`, even though the authentication didn't check-out.

Variants

1. Overwrite the stack-frame's saved return address to return to data that has been overwritten using the overflow with code to invoke the shell (via the `execve` system call) and give the attacker access to running what she wants
2. Overwrite the header created by `malloc`
3. Overflow the `int` passed to malloc
4. Overwrite the program's *secret key* with a known key, so that future encrypted messages can be decrypted
5. Modify state variables, like in the example above where `authenticated` was overwritten
6. Server programs might communicate with a database with SQL strings, which we can overwrite
7. If you don't know exactly where to jump to, you can create a nop sled, where processor will jump somewhere into your sled and slide into your code
8. "Read overflow" --- *read* instead of write beyond the buffer
    1. You ask client for both length of message and message, and you print chars up to that length. But if the length was longer than the message,

    then you'll print chars *beyond* the message, and they can read it.
    2. E.g. **Heartbleed**
    3. With this they can get passwords and crypto-keys
9. "Format string"

   with the code

   ```
   char *mystring = "%d %x";
   printf(mystring);
   ```

   `printf` will try to find the variables corresponding to those values off the stack frame, but they don't exist, so it is reading someone else's memory.

   The safe way to do it is like

   ```
   printf("%s", mystring);
   ```

   because now it print the string literal `"%d %x"` without trying to interpret it.

10. Note that you get total **memory safety** by simply *not* using C or C++

Protections

1. **Stack canary** --- save a number at a known location on the stack, and make sure it doesn't get overwritten during the function, else abort
2. Make stack (and heap) non-executable
    1. But we can still jump to code that calls `execve` *inside* `libc`, which is known to be in our process's address space
    2. So we can use **Address-space Layout Randomization** to make it hard to find `libc`
    3. But this is defeated using "gadgets" and "blind ROPs"
    4. We can defeat "blind ROPs" with "Control-flow Integrity" (which isn't used currently) which defines the program's "expected behavior" and detects deviations from that *efficiently*

Secure coding in C[++]

1. Follow the "CERT C Coding Standard"
    1. There are similar guides for other languages
2. Use techniques like static program analysis, fuzz testing, and symbolic execution
3. Make sure the length the user gave you is not longer than their actual input
4. Validate input
5. Put pessimistic checks on assumed preconditions
6. Use safe versions of the string functions
    1. E.g. `strcpy(buf, "str")` becomes `strlcpy(buf, "str", sizeof(buf))`

    2. And `strcat` becomes `strlcat`
7. Don't forget about the byte added by the NUL terminator
8. Get pointer arithmetic right, e.g. (`int + 1`) adds 4 bytes
9. Don't reuse memory malloc'd before by setting `p` to `NULL` after passing it to `free(p)`
10. Use `goto` chains to properly free memory when you exit early from a function
    1. I think they did this all the time in the Linux kernel
11. Use safe libraries
    1. the Very Secure FTP string library or `std::string` which don't rely on null-terminators to find the length of strings
    2. Use smart pointers (standard in C++11)
    3. For networking use Google's *protocol buffers* or *Apache Thrift*
12. Use a safe allocator

Problems on the WWW

- *Common theme:* **validate your input**

SQL Injection

The Attack

E.g. when you enter your name into a form as

```
someone' OR 1=1); DROP TABLE mytable; --
```

so that PHP creates a SQL query

```
SELECT * FROM mytable WHERE (name='someone' OR 1=1);
DROP TABLE mytable;
-- their code is commented out;
```

which retrieves all data then deletes it.

Protections

1. **Sanitize** the input --- deleting or escape characters we don't want, like `' ; --`
2. Stronger, **whitelist** --- make sure that it matches something that is valid before appending it to the query string.
3. **Limit privileges** that this instance of a connection to the database has, so it doesn't have the right to e.g. drop the database.
4. **Encrypt sensitive data** in the database, so when they get it they can't read it.

Session Hijacking

1. Servers send clients *cookies* which they send to the server on subsequent requests
2. Servers know requests with your cookies came from you, and that way you can access your private data on the server
3. So by stealing a cooking, an attacker can impersonate a user
4. Protect against this by sending session-cookies over HTTPS
5. Delete 'logged-in' cookies after session ends
6. Have them 'time out' anyway after some long period of time

Cross-site Request Forgery (CSRF)

Example of Mechanism

1. A user is logged in on `bank.com`, and has appropriate session-cookies
2. The user navigates to `attacker.com/index.html`
3. `index.html` has `<img src="bank.com/transfer?amt=9999&to=attacker">`
4. The user's browser automatically issues a `GET` to the page in the `img` tag
5. So the bank sees an authenticated transfer request to the attacker and performs it

User could also be tricked into clicking a link for the `GET` request, e.g. from a spam email.

Protections (on `bank.com`)

1. Ensure that the `GET` header's `Referer` field is within `bank.com`
2. Only respond to requests that contain a hidden form field in every link & form that hacker has trouble guessing (e.g. the user's session ID)

Cross-site Scripting (XSS)

Same Origin Policy (SOP)

1. Browser associates web page elements (layout, cookies, events) with a given *origin* (e.g. `bank.com`)
2. Only scripts received from a web-page's origin have access to that page's elements (layout, cookies, events)
3. XSS is a way to circumvent this

XSS Mechanisms

1. **Stored/Persistent XSS** --- attacker plants script to transfer them cache directly on the `bank.com` server
    1. This was used to take MySpace down for a weekend
2. **Reflected XSS attack**
    1. Attacker gets you to send `bank.com` its Javascript (via a URL)

   2. `bank.com` "echoes" that Javascript back to you
   3. Your browser executes the script, within the `bank.com` origin
   4. **Defense** --- `bank.com` ought to *sanitize* its responses to not contain `script` tags sent by the user

Passwords

5/1/14

References:

1. Coda Hale: How to safely store a password
2. Security Stack Exchange: How to securely hash passwords
3. OWASP: Password Storage Cheat Sheet

Recurrent theme

> Don't write your own code for this stuff

Why bcrypt?

- It is *slow* to compute, unlike MD5, SHA-X
- *Salts* don't prevent **dictionary attacks** *(see Glossary above)* or brute force attacks
- It has a parameter to allow you to choose the compute requirements
- It has salts built-in to prevent **rainbow table** attacks

Salting

- The salt is a **128-char string**
- You save `"salt:HASH(salt+password)"` into the database
    - Note the salt is visible there as plaintext, this is OK as long as you're not just *giving these away*
- This means no one can use a table of pre-hashed strings to attack you
- They *can* find your salt, but then they'd have to compute all the hashes *for each user*

Some Don'ts

Paraphrased from stackexchange

> **Complexity is bad. Homemade is bad. New is bad.** If you remember that, you'll avoid 99% of problems related to password hashing, or cryptography, or even security in general. Do *not* write a tangled mess of operations and expect that to confuse attackers.

Some Dos

- Use as many iterations as you can tolerate on your server
- Use `/dev/urandom` to obtain salts. **Don't** use the *username*.

Cryptographic Hash Functions

Hashing Vs. Encryption

You can't un-hash, but you can *decrypt*

Message-Digest

- **MD5** --- *has had some problems*; produces 128-bit / 16-byte / 32-digit- hex hash value
    - It's not *that* secure, it seems like
- **SHA-1** --- produces 160-bit / 20-byte / 40-digit-hex
    - **Uses** --- encryption (not so secure), data integrity (e.g. Git)
- **SHA-2** --- more secure than SHA-1, really secure.

Other Noteworthy Algorithms

- **AES** (Advanced Encryption Standard) --- **symmetric** and popular within the U.S. government
- **Base64** -- encode into alphanumeric+2
    - The last 2 chars might be `[(+,/),(+,-),(-,_),(.,-),(_,:),(!,-), etc.]`

RSA

Rivest, Shamir, Adleman --- **deterministic, asymmetric** cryptosystem widely used for secure data transmission

- *Public* encryption key, *Private* decryption key
- Based on the difficulty of factoring the product of two large prime numbers
- Public key --- the product of two large prime numbers and an auxiliary value
- Private key --- the two large prime numbers
- **Operation**
    - Choose two *distinct* prime numbers *p* and *q* (at random, of similar bit-length)
    - Compute $n = pq$ and $\Phi(n) = (p-1)(q-1)$
    - Choose an integer e s.t. $1 < e < \Phi(n)$, and also e & $\Phi(n)$ are *coprime* (share no prime factors)
        - *e* is the *public key exponent*
        - smaller *e* will be more efficient to compute, but slightly less secure

    - *d* := multiplicative inverse of *e* (modulo $\Phi(n)$) [sic]
        - *d* is the *private key exponent*
    - **Encryption**:
        1. Alice sends public key (n, e) to Bob
        2. Bob turns message M into an integer m s.t. $0 \leq m < n$ using an agreed-upon reversible **padding scheme** protocol
        3. Bob computes the ciphertext $c \equiv m^e$ (mod n) and transmits c to Alice
    - **Decryption**:
        1. Alice recovers m from c by computing $m \equiv c^d; (\text{mod}; n)$
        2. Now she reverses the padding scheme to recover M from m
    - **Signing Messages**: note that Bob (the public key holder) can **authenticate** messages Alice **signs** using her private key. This means that Bob can verify both that Alice sent it, and that the message hasn't been tampered with in the process.
        - This is *key*
- One can avoid a lot of the known methods of attack by embedding structured, randomized padding into the value m before encrypting it
    - Padding is very difficult, so there is a patent-expired secure padding scheme known as **RSA-PSS**
    - Padding schemes "are as essential for the security of message signing as they are for message encryption"
- Integer factorization and the RSA problem
    - For $\text{len}(n) \leq 300$, n can be factored in a few hours on a PC using free software
    - For $\text{len}(n) \geq 2048$, you should be safe for a while
    - Shor's algorithm shows that a quantum computer would be able to factor in polynomial time, breaking RSA

Notes about Practice

Ref: Apple

Asymmetric encryption is often used for establishing a shared communication channel. Because asymmetric encryption is computationally expensive, the two endpoints often use asymmetric encryption to exchange a symmetric key, and then use a much faster symmetric encryption algorithm for encrypting and decrypting the actual data.

Asymmetric encryption can also be used to establish trust. By encrypting information with your private key, someone else can read that information with your public key and be certain that it was encrypted by you.

Typical use of Public Key, and Reason it is Naive

1. Alice generate public/private key pair
2. Alice sends the public key to Bob, unencrypted
3. Alice encrypts her message (or any portion of the message) using her private key and sends it to Bob
4. Bob decrypts it with Alice’s public key, proving it came from Alice
5. Bob encrypts his message using Alice’s public key and sends it to Alice.
6. Alice decrypts the message with her private key.

Problem:

The authentication method of encrypting the message with your private key can be got around by a man-in-the-middle attack, in which someone with malicious intent ("Eve") replaces Alice’s original message with her own, so that Bob Eve’s instead. Eve then intercepts & alters each of Alice’s messages. When Bob receives the message, he decrypts it with Eve’s public key, thinking that the key came from Alice.

HTTPS[ecure]

> *The first step in providing secure services is using HTTP Secure (HTTPS), which encrypts your packets.*

Overview

1. Client and server exchange an *SSL certificate* in a `.crt` file ($10-300)
2. Now client *authenticates* server (using Web of Trust, scariest part of the process)
3. Client uses the server's public key to send him an encrypted secret
4. Server decrypts this secret using his *private* key (which *only* the authenticated server can do!)
5. Both the client and the server use the secret to locally generate the session symmetric key
6. Now they can talk safely because no one else knows their session symmetric key
    - However, a third party with access to the shared secret will be able to generate the session symmetric key and decrypt the communication.

Cryptography

XOR

XOR is represented by the symbol "$\oplus$"

Properties:

$$x \oplus x \equiv 0$$

$$x \oplus 0 \equiv x$$

Associativity & commutativity

From these we can derive that

$$(x \oplus y) \oplus x = (x \oplus x) \oplus y = 0 \oplus y = y$$