(Only the most basic) Encryption in a Nutshell
Defensive Coding

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://www.cs.stevens.edu/~jschauma/631/
Cryptography

Cryptography can provide “security” in the areas of:

- Authenticity
  - *Is the party I’m talking to actually who I think it is?*

- Accuracy or Integrity
  - *Is the message I received in fact what was sent?*

- Secrecy or Confidentiality
  - *Did/could anybody else see (parts of) the message?*
How does encryption work?

*Secrecy:* Make sure that the data can only be read by those intended.
How does encryption work?

Secrecy: Make sure that the data can only be read by those intended.

- Alice and Bob agree on a way to transform data
- transformed data is sent over insecure channel
- Alice and Bob are able to get data out of the transformation
How does encryption work?

Different approaches:

- public key cryptography
- secret key cryptography
How does encryption work?

Different approaches:

- public key cryptography (example: RSA, your ssh keys)
  
  - Alice has a private and a public key
How does encryption work?

Different approaches:
- public key cryptography (example: RSA, your ssh keys)
  - Alice has a private and a public key
  - data encrypted with her private key can only be decrypted by her public key and vice versa

```
Bob
| Hello Alice! | Encrypt | Alice's public key |
```

```
Alice
| Hello Alice! | Decrypt | Alice's private key |
```

Lecture 11: Encryption in a Nutshell, Defensive Coding
November 27, 2017
How does encryption work?

Different approaches:

- public key cryptography (example: RSA, your ssh keys)
  - Alice has a private and a public key
  - data encrypted with her private key can only be decrypted by her public key and vice versa
  - shared secrets derived from public key cryptography can be used as e.g. symmetric session keys
How does encryption work?

Different approaches:

- secret key cryptography (example: *AES*)
  - Alice and Bob share a secret key
  - for authentication purposes, Alice may prove to Bob that she knows the secret key
  - any data encrypted with this key can also be decrypted using the same key
Cipher Modes

Encryption entails transformation of input data ("plain" or "clear" text) into encrypted output data ("ciphertext"). Input data is generally transformed in one of two ways:

*Stream Cipher*: each bit of plaintext is combined with a pseudo-random cipher digit stream (or *keystream*)

*Block Cipher*: fixed-length blocks of plaintext are transformed into same-sized blocks of ciphertext; may require padding
Electronic Codebook Mode

Electronic Codebook (ECB) mode encryption

Electronic Codebook (ECB) mode decryption
Electronic Codebook Mode
Cipher Block Chaining

Cipher Block Chaining (CBC) mode encryption

Cipher Block Chaining (CBC) mode decryption
Using Crypto

- don’t write your own crypto code, use existing libraries
- don’t invent your own security protocol, even if you can’t think of a way that you could break it
- don’t invent your own source of entropy
- always seed your PRNG, salt your hashes
- default to reasonable crypto primitives:
  - 2048 bit RSA for asymmetric key cryptography
  - AES256-CBC for symmetric key cryptography
  - HMAC-SHA256 for integrity
Random String generation

Random numbers can be generated using `/dev/random`, `/dev/urandom`, `rand(3)`, `random(3)`, `BN_rand(3)` etc.

Map numbers to printable characters (for use as a salt, for example):

```c
static const unsigned char itoa64[] = 
    "/0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
```

```c
char salt[16];
for (i=0; i<16; i++)
    salt[i] = itoa64[(int)random()%64];
```

See also: `rand.c` from Lecture 11.
On being random...

```c
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}
```
On being random...

"Sony uses a private key, usually stored in a vault at the company's HQ, to mark firmware as valid and unmodified, and the PS3 only needs a public key to verify that the signature came from Sony," he said. "Applied correctly, it would take billions of years to derive the private key from the public key, or to make a signature without knowing the private key, even when you have all the computational power in the world at your disposal. The signing recipe requires that a random number be used as part of the calculation, with the caveat that that number must be truly random and not predictable in any way. However, Sony wrote their own signing software, which used a constant number for each signature."

https://youtu.be/LP1t_pzxKyE
https://is.gd/7Dmtwv
On being random...

```
$ cc -Wall prng.c
$ ./a.out
    16807   692375228   699350133
$ ./a.out
    16807   692476070   699501396
$ ./a.out ; echo ; ./a.out
    16807   692593719   699652659
    16807   692593719   699669466
```
Handling Secrets

Never hardcode secrets in your code.

Here’s how the hack went down: Two attackers accessed a private GitHub coding site used by Uber software engineers and then used login credentials they obtained there to access data stored on an Amazon Web Services account that handled computing tasks for the company. From there, the hackers discovered an archive of rider and driver information. Later, they emailed Uber asking for money, according to the company.

https://is.gd/qRe7Z5
https://is.gd/ZaJDS7
Handling Secrets

- use a Key Management System; integrate with common libraries/API
- allow the user different options of providing secrets; see e.g. `openssl(1)`
  - on the command-line (note: visible in process table!)
  - via the environment (note: possibly visible to other users; often then stored in shell initialization files)
  - from a file (note: ensure correct permissions!)
  - from a file descriptor
  - from stdin
  - prompt from the tty
- sanitize / zero out secrets after use
- don’t log secrets!
Handling Secrets

$ cc -Wall getpass.c
$ echo foo | ./a.out
$ echo foo | env SECRET=password ./a.out
$ ./a.out -p password
^Z
$ ps wwaux | grep a.out
Input validation

All data is tainted until proven otherwise.
Input validation

Never trust anything from outside of your control. This includes:

- data input directly provided by the user
- data indirectly / implicitly controlled by the user (e.g. HTTP headers)
- data read from files you think you control (e.g. config or state files)
- anything from the environment;
  - use `getpwent(3)` instead of e.g. HOME or USER
  - explicitly set e.g. PATH, LD_LIBRARY_PATH
  - explicitly unset e.g. LD_PRELOAD
Input validation

- length checks (in both directions!)
- range checks on numeric fields, character ranges
- check path names against directory escapes (./../../)
- prefer whitelists over blacklists
- encode data before validation or use
- use type check assertions
Type Checks

Prefer "is a" tests over "looks like";

IPv4:

if (inet_pton(AF_INET, $ip)) {
    // AF_INET
} elseif (inet_pton(AF_INET6, $ip)) {
    // AF_INET6
} else {
    // not an IP address
}
Subprocesses

- don't use `system(3), popen(3)` with any user provided input
- prefer `fork(2)/exec(3)`
- explicitly set a trusted `PATH, LD_*` etc.
- never invoke commands from a temporary or relative location (e.g. `/tmp/cmd, ./cmd`)
- set a suitable `umask(2)`
Setuid

- drop privileges as early as possible
- only raise privileges for sections you need
- permanently drop privileges if you no longer need them
- be aware of which subprocesses might let you break out of your program or which could spawn a shell (e.g. `vi(1)`)  
- be aware of which operations are atomic and which aren’t
- beware signal and exit handlers
File I/O

- avoid wherever possible
- assert suitable protection on private files (see e.g. \texttt{ssh(1)})
- be careful when opening, unlinking, overwriting files based on user input / user provided pathnames
- don’t use temporary files
  - set a restrictive \texttt{umask(2)}
  - use \texttt{mktemp(3)}
  - unlink via exit handler

https://www.netmeister.org/blog/mktemp.html
Coding style

Always use braces.

```c
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
               uint8_t *signature, UInt16 signatureLen)
{
    OSStatus err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

    fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

https://www.imperialviolet.org/2014/02/22/applebug.html
Coding techniques

- treat functions as black boxes, minimize side-effects, avoid global variables
- explicitly mark variables / function arguments as ‘const’ (see also: const.c)
- check your return codes!
- avoid magic numbers
- use `strncpy(3)/strncat(3)` etc. instead of `strcpy(3)/strcat(3)` etc.
- fail early, fail explicitly
- allocate / free resource in same scope
- check the boundaries of your buffers
- use compiler options (e.g. `-fsanitize=address`), debugging libraries, analysis tools (e.g. valgrind)
- understand and resolve all compiler warnings
  (`-Wall -Werror -Wpedantic`)
Use-after-free

Hi, we found a use-after-free vulnerability which is exploitable to RCE in the SMTP server.

According to receive.c:1783,
1783 if (!store_extend(next->text, oldsize, header_size))
1784 {
1785     uchar *newtext = store_get(header_size);
1786     memcpy(newtext, next->text, ptr);
1787     store_release(next->text);
1788     next->text = newtext;
1789 }

when the buffer used to parse header is not big enough, exim tries to extend the
next->text with store_extend function. If there is any other allocation between
the allocation and extension of this buffer, store_extend fails.

store.c
276 if (((char *)ptr + rounded_oldsizel = (char *)(next_yield[store_pool]) ||
277     inc > yield_length[store_pool] + rounded_oldsizel - oldsize)
278     return FALSE;

Then exim calls store_get, and store_get cut the current_block directly.
store.c
208 next_yield[store_pool] = (void *)((char *)next_yield[store_pool] + size);
209 yield_length[store_pool] -= size;
210
211 return store_last_get(store_pool);

However, in receive.c:1787, store_release frees the whole block, leaving the new
pointer points to a freed location. Any further usage of this buffer leads to a
use-after-free vulnerability.

To trigger this bug, BDAT command is necessary to perform an allocation by raising
an error. Through our research, we confirm that this vulnerability can be exploited
to remote code execution if the binary is not compiled with PIE.

https://bugs.exim.org/show_bug.cgi?id=2199
Core Principles

- Simplify – don’t write any code you don’t need.
- Minimize your Attack Surface – only expose (interfaces, API functionality, access, ...) what is needed
- Secure Defaults – user and group permissions, umask, PATH, locations, ...
- Assume that Human Behavior Will Introduce Vulnerabilities into Your System
- Know Your Enemy – understand your threat model
Core Principles

- Principle of Least Privilege – only access, use, or accept information/resources that are strictly needed; don’t run unprivileged unless/until privileged mode is needed
- Fail Closed – (unexpected) failure must not lead to e.g. access, information disclosure, increased privileges, ...
- Defense in Depth – any component or tool needs to be safe to use; do not rely on outside mechanisms or protections
- Kerkhoff’s Principle – ”the enemy knows the system”; avoid Security by Obscurity
- Assume a Hostile Environment
  - always use transport encryption
  - always authenticate all parties
  - authentication ≠ authorization
References and Links

Crypto:
- `crypto(3)`
- `EVP_EncryptInit(3)`
- `EVP_BytesToKey(3)`
- [http://tldp.org/LDP/LG/issue87/vinayak.html](http://tldp.org/LDP/LG/issue87/vinayak.html)
- [https://is.gd/khpsCT](https://is.gd/khpsCT)
References and Links

Defensive Coding

- https://is.gd/ChD04C
- https://is.gd/ee4nXr
- https://www.owasp.org/index.php/Data_Validation
- https://www.netmeister.org/blog/mktemp.html
- https://is.gd/JVS5lI
- https://is.gd/QMru7v
References and Links

Bugs and exploits

- https://bugs.exim.org/show_bug.cgi?id=2199
- https://is.gd/7Dmtwv
- https://is.gd/qRe7Z5
- https://is.gd/ZaJDsT
- https://www.imperialviolet.org/2014/02/22/applebug.html