Lec I Introduction.

CIA

Lec 2 Security Principle

- Don't Blame the User
- Security is the economics
- Countermeasures:

1° Prevention:

Ex. bufferoverflow ← Memory-safe language python

— prevention fails, fail hard

Ex: Bitcoin transfer – irreversable

2° Detection & Defense in depth:

— False positive: alert when nothing there → cost

— False negative: fail to alert when something there → failure

— defense in depth:

Ex. compose detectors — 2 independent: FP1, FP2 & FN1, FN2

① parallel composition: either may alert

FN: $FN_1 \times FN_2$ FP: $FP_1 + FP_2$

② serial composition: both alert → trade off

FN: $FN_1 + FN_2$ FP: $FP_1 \times FP_2$

Ex: Password Authentication. — reuse password.

1° password manager

2° two factor authentication: need both correct password & separate device

- Measure Attacker Capabilities
  - — time & capabilities needed for an attacker
  - — security is economics < security ↑, cost ↑ / purchasers ↑, cost ↓
- Least Privilege
- Design Patterns for Building Secure Systems
  - —— ☆ The Trusted Computing Base (TCB)

1° identify TCB

2° security requirement of TCB:

① correct ? → verify?

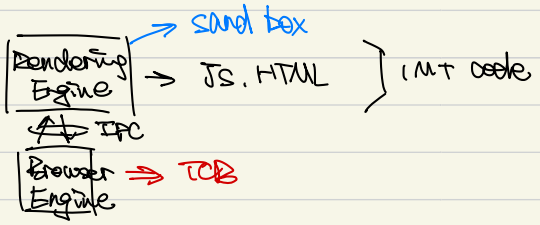
② complete ? → bypassable?

③ tamper-resistant ? → integrity of the TCB must be maintained.

☆ KISS: keep it simple, stupid!

Ex: Web Browser

separate privilege



Benefits of TCBs:

i) system separation

ii) modularity ⇒ isolation. keep potential problems localised.

- Ensuring complete mediation

✡ TCB component: reference monitor → access control

  - verifiable
  - un-bypassable
  - tamper-proof

  - single point through which all access must occur

- use fail-safe defaults

  Ex: firewall – when fail, no packets can pass

- consider human factors
- only as secure as the weakest link
- don't rely on security through obscurity
- trusted path: trusted way to verify
- TOCTTOU: Time of check to (race condition) time of use vulnerability

✡ reduce size of TCB

i) privilege separation

ii) separation of responsibility

Lec 3. Buffer Overflows

1° where user input goes
2° conditions in ifs & loops
3° unsigned vs. signed
4° printf & format string vulnerabilities

Ex:

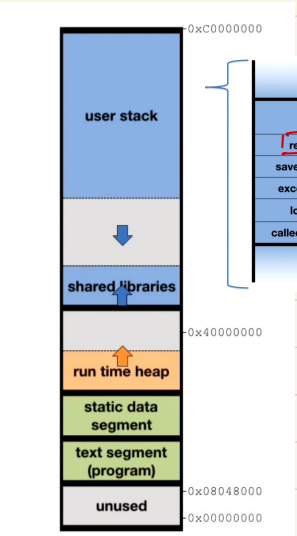
```
char name[20];
int (*fptr)();

void vulnerable() {
    ...
    gets(name);
    ...
}
```

- when malicious code > 20 bytes:

input 20 bytes of garbage + addr of the attacker's code + attacker's code

1° stack smashing: overwrite the return address



2° signed / unsigned vulnerability

Ex:

```
void f(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

size_t

memcpy(void *s1, const void *s2, size_t n);

3° Integer Overflow vulnerability

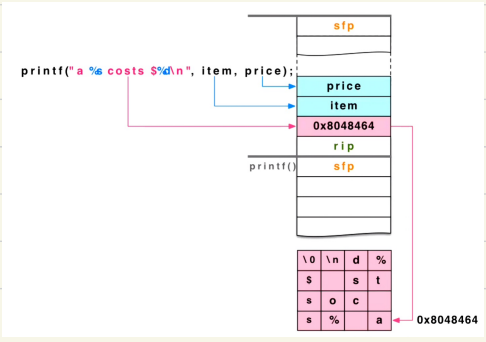
0x FFFFFFFF

Ex.

```
void f(size_t len, char *data) {
    char *buf = malloc(len + 2);
    if (buf == NULL) return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len+1] = '\0';
}
```

☆ %N 的參數是地址.

4° Format String Vulnerability




```
printf("100% dude!");
    ⇒ prints value 4 bytes above retaddr as integer
printf("100% sir!");
    ⇒ prints bytes pointed to by that stack entry
      up through first NUL
printf("%d %d %d %d ...");
    ⇒ prints series of stack entries as integers
printf("%d %s");
    ⇒ prints value 4 bytes above retaddr plus bytes
      pointed to by preceding stack entry
printf("100% nuke'm!");
```

%n → writes the number of characters so far into the corresponding format argument

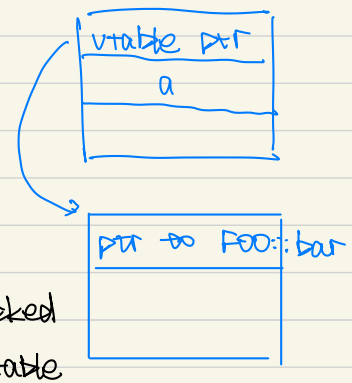
⇒ random address write ! ☆

5° Vulnerabilities Outside the Stack

— heap Overflow : overwrite a vtable pointer

C++ :

```
class Foo {
    int a;
    public virtual void bar() {...}
    ...
```



Def:

A buffer in the heap is not checked : write beyonds and overwrites the vtable ptr of the next object in the heap

— Use after free

- write data to a freed buffer which might be allocated to another object.

■ Memory Safety : No accesses to undefined memory

← read access
write access
execute access

① Spatial Safety:
No access out of bounds

② Temporal Safety:
No access before or after lifetime of object.

Lec 4 Buffer Overflow Defenses

- memory-safe language
- non-memory-safe language → bounds check
- OS / compiler level → harden non-memory-safe language code against exploits

1° Reasoning about memory safety

— module by module basis:

precondition: what must hold for function to operate correctly

postcondition: what holds after function complete.

Ex: /* p can't be NULL and p a valid pointer */ ← precondition

①
```
int deref (int *p) {
    return *p;
}
```

② postcondition: p is not NULL and valid

```
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

③ a != NULL & n <= sizeof(a)

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

a != NULL && i < size(a)

invariant: i < n && n <= size(a)

① identify each point of memory access

② write down precondition it requires

③ propagate requirement up to func level

④

```
char *tbl[N]; /* N > 0, has type int */

int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

size_t

size_t

size_t

postcondition:

⇒ integer overflow:

if h<0, h%N<0

memory unsafe

same for every function.

2° Code Hardening

① STACK Canaries: random values chosen when program starts

goal: protect return address

— store canary before saved return values

— function prologue pushes canary, epilogue checks canary against stored value to see if it has changed.

ATTACK stack canary:

① • Learn the value of the canary, and overwrite it with itself
  • e.g., a format string vulnerability, an information leak elsewhere that dumps it

② • Random-access write past the canary ⇒ format string vulnerability
  • Canary only defends against consecutive writes

③ • Overflow in the heap
• Overwrite function pointer or C++ object on the stack
• Bottom line: Bypassable but raises the bar
  • A simple stack overflow doesn't work anymore: Need something a bit more robust
  • Minor but nearly negligible performance impact

② Non-Executable Page (aka DEP, W^X)

— mark writeable pages as non-executable
executable non-writable

- No noticeable performance impact
- can still overwrite local variables

ATTACK:

return to libc: set up the stack and "return" to execl

Return Oriented Programming

- Idea: chain together "return-to-libc" idea many times
  - Find a set of short code fragments (gadgets) that when called in sequence execute the desired function
  - Inject into memory a sequence of saved "return addresses" that will invoke them
  - Sample gadget: add one to EAX, then return
- ROP compiler
  - Find enough gadgets scattered around existing code that they're Turing-complete
  - Compile your malicious payload to a sequence of these gadgets

against.

③ Address Space Layout Randomization (ASLR)

- Randomize the start of different memory sections.
  - code
  - library
  - start of the stack & heap.

⇒ return address modification unknown

- Performance overhead is close to 0%
- ASLR + DEP ☆ defense in depth
  - need both buffer overflow and a separate information leak (way to fill in the return addresses for ROP chain)

4° Fuzz testing

Program crashes ⇒ bug

— generate test cases:

- Random testing: generate random inputs
- Mutation testing: start from valid inputs, randomly flip bits in them
- Coverage-guided mutation testing: start from valid input, flip bits, measure coverage of each modification, keep any inputs that covered new code

□

strcpy(,) ⟵ strncpy

gets ( ) ⟵ fgets( )

□

```
void foo (int *dest) {
    printf ("Hello world! %on\n", dest);
}
```

通过控制指针 ⇒ 随机地址写

%n 将打印字符串.