Memory Vulnerabilities: Attacks and Solutions

Assumptions in system attacks

- **Exploit:** Malware programmers find an exploit that is on many vulnerable machines
  - Malware makes use of memory bugs in a process
- **Replicable:** Same code runs on many machines, with the same starting address for the stack
  - Same malware code succeeds on all vulnerable machines with no additional effort

Buffer overflow attacks

- **The unsafe languages problem**
  - Use memory vulnerabilities often caused by programming in unsafe languages
- **Programming problems**
  - Poor handling of pointers to memory locations can cause such attacks
- **Benefit**
  - Gives remote attacking capabilities

Basic idea of buffer overflow

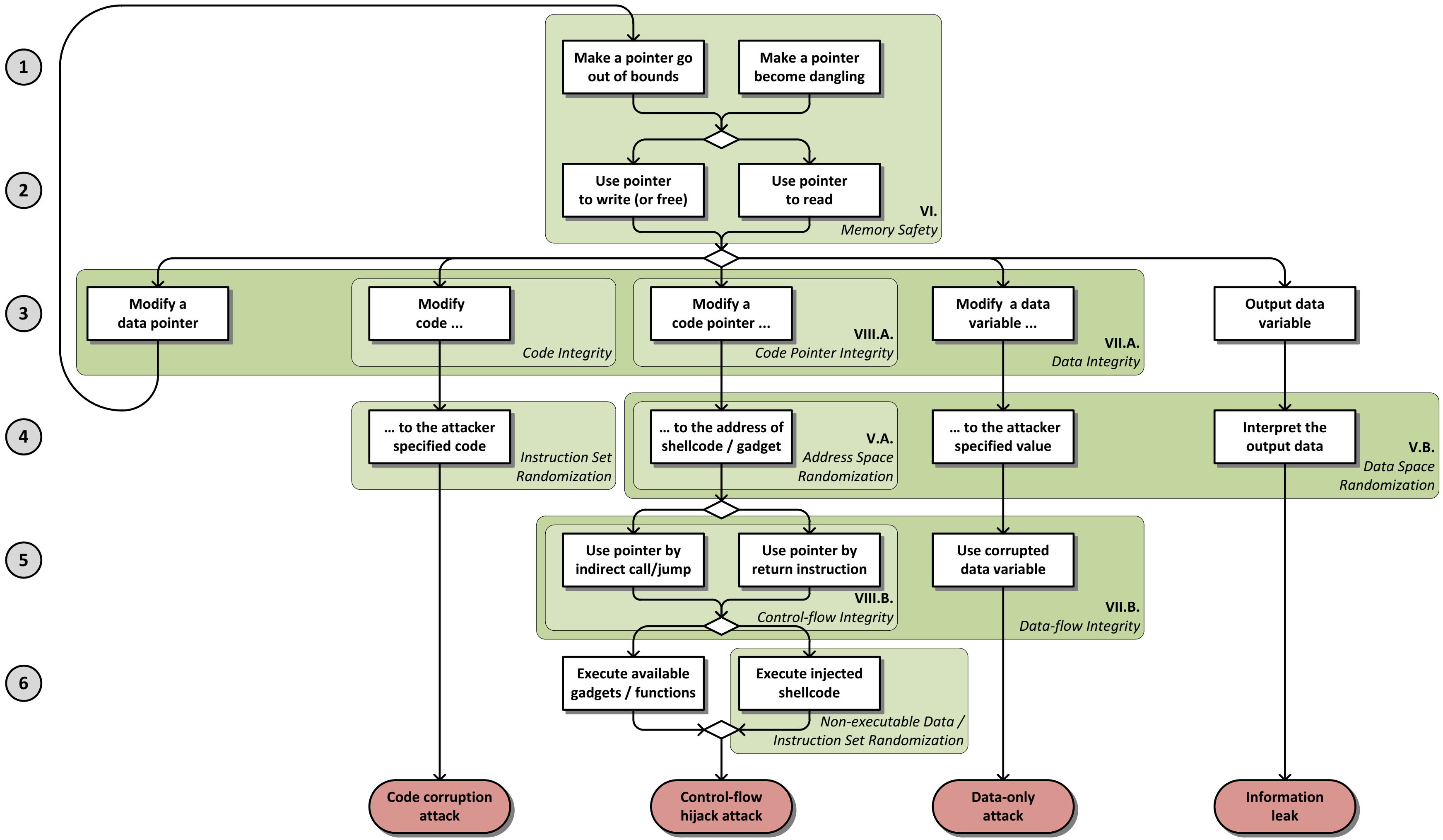
- **Find a memory vulnerability**
  - such as improper boundary checks on a C array.
- **Inject code**
  - into the victim program's memory.
  - most programs include buffers for their operations.
  - Buffer can be on stack, heap, or in a static area.
- **Misuse existing code**
  - In some cases, only jump to existing code

Pointer dereferencing attacks

- Make the pointer invalid, then dereference it.
- Spatial error: dereference dangling out-of-bounds pointer (out of bounds of the object)
- Temporal error (use-after-free): dereference dangling pointer (pointing to deleted object)
  - Area need to be reused by another object.

Buffer overflow

- INDEXING BUG: Incrementing or decrementing an array pointer in a loop without proper bound checking
- Integer overflow helps in exploiting indexing bugs (truncation, signedness, incorrect casting)
- Exploiting an incorrect exception handler, deallocates an object, does not reinitialize the pointers to it



Reference: Szekeres et al. 2013

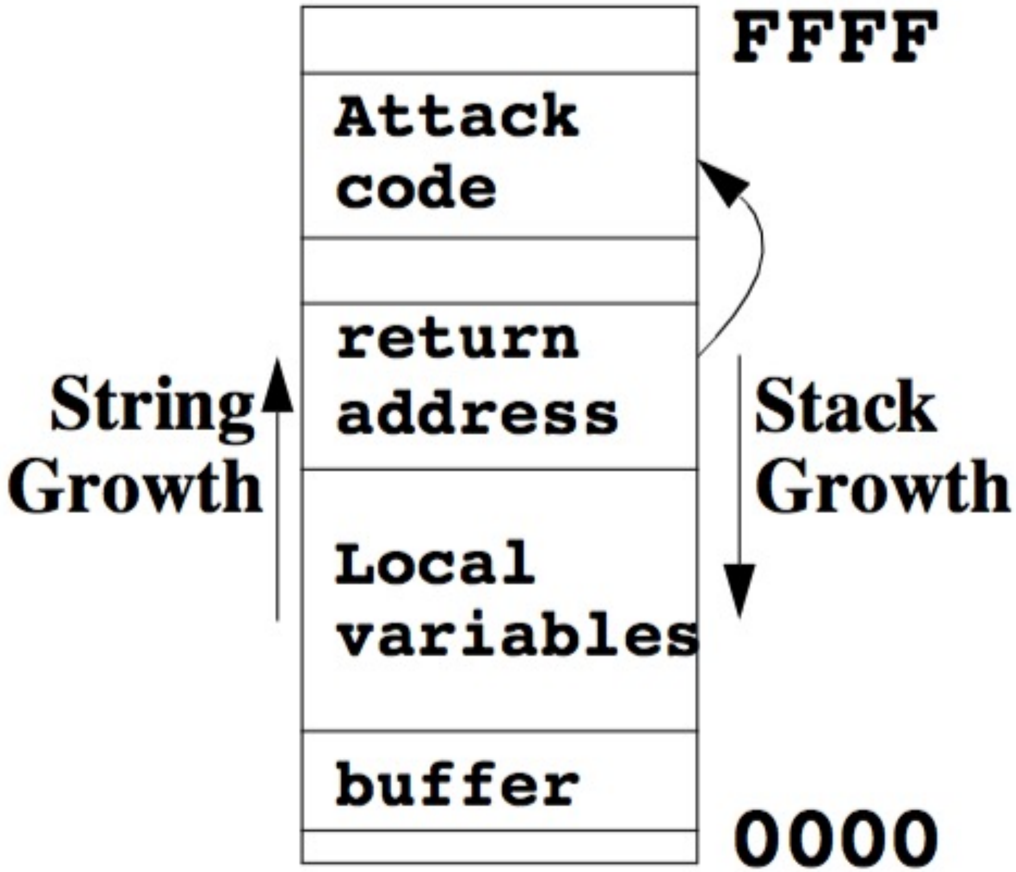
More exploits

- **Function pointers**
  - Corrupt a pointer to a function by an adjacent buffer:
  - `void (* foo)()`
- **Longjmp buffers**
  - Instead of longjmp to programmer's buffer, go to the location of the malicious code

Stack smashing attack

- Program routines store parameters and automatic variables as well as return address on the stack.
- Attackers try to corrupt automatic variables to
- write malicious values on the adjacent memory locations => overwriting the return address.

Stack smashing attack



Exploiting the jmp instruction

- Insert a `jmp` instruction at an appropriate place to execute the injected code.
- Optionally, `jmp` to exec and load full malware binary into memory as an independent process.
  - In this case, the attacker only needs to modify the pointer to string as the argument for exec and cause the program to jump to it. No attacker injection needed.

Example

Corrupt the memory locations in the table.

```
func_ptr jump_table[3] = {fn_0, fn_1, fn_2};
jump_table[user_input]();
```

Attack techniques

- Format-string attacks
  - Location of to be printed data is provided to printf
    - `printf(str);`
  - Attacker controls the location, loads payload into location
  - Relies on absolute address of the location

Attack techniques

- Data modification attacks
  - A shell command adjacent to end of a vulnerable buffer
  - Attacker attempts to override the command
  - Succeeds depending on the relative distance between command and buffer

Attack techniques

- Heap overflow
  - Using vulnerable buffer to corrupt heap allocated blocks
  - Use corrupted blocks to inject code
- Double-free attacks
  - block = malloc(), free(block), free(block)
  - Result: attacker can inject arbitrary code in block
- Integer overflow attacks
  - Limited memory capacity causes integers to change value due to overflow
  - Checks based on integer value vulnerable to this attack

Attack Example: gets

```
#include <stdio.h>
int main () {
    char username[8];
    int allow = 0;
    printf("Enter your username, please: ");
    gets(username); // user inputs "malicious"
    if (grantAccess(username)) {
        allow = 1;
    }
    //has been overwritten overflowing char username[8]
    if (allow != 0) {

        privilegedAction();
    }
    return 0;
}
```

https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml

Attack Mitigation Example: gets

```
int main () {
    char* username, *nlptr;
    int allow = 0;

    username = malloc(LENGTH * sizeof(*username));
    if (!username)
        return EXIT_FAILURE;
    fgets(username,LENGTH, stdin);

    if (grantAccess(username)) {
        allow = 1;
    }
    if (allow != 0) {
        priviledgedAction();
    }
    free(username);
    return 0;
}
```

Attack Example: strcpy

```
char str1[10];
char str2[]="abcdefghijklmn";
strcpy(str1,str2);
```

Use definite size strings (strncpy)

```
char str1[BUFFER_SIZE];
char str2[]="abcdefghijklmn";

strncpy(str1,str2, BUFFER_SIZE); /* limit number of characters
to be copied */
```

Attack Mitigation Example: strcpy

```
#ifndef strlcpy
#define strlcpy(dst,src,sz) snprintf((dst), (sz), "%s", (src))
#endif


int buffer_length = strlcpy(dst, src, BUFFER_SIZE);

    if (buffer_length >= BUFFER_SIZE) {
        print(“String too long”);
    }
```

strlcpy is available as a library function on BSD systems

Attack Example: sprintf

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

enum { BUFFER_SIZE = 10 };

int main() {
    char buffer[BUFFER_SIZE];
    int check = 0;

    sprintf(buffer, "%s", "This string is too
long!!!!!!!!!!");

    printf("check: %d", check);
       /* This will not print 0! */

    return EXIT_SUCCESS;
}
```

> gcc sprintf-of.c
> ./a.out
> * stack smashing detected ***: ./a.out terminated**
> check: 0Aborted (core dumped)

> gcc sprintf-of.c -fno-stack-protector
> ./a.out
> Segmentation fault (core dumped)

Attack Mitigation Example: sprintf

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

enum { BUFFER_SIZE = 10 };

int main() {
    char buffer[BUFFER_SIZE];

    int length = snprintf(buffer, BUFFER_SIZE, "%s%s", "long-name", "suffix");

    if (length >= BUFFER_SIZE) {
        /* handle string truncation! */
    }

    return EXIT_SUCCESS;
}
```

Attack Example: printf

```
#include <stdio.h>

int main(int argc, char **argv) {
    char *secret = "This is a secret!\n";

    printf(argv[1]);

    return 0;
}
```

```
warning: format string is not a string
literal (potentially insecure) [-Wformat-
security]
    printf(argv[1]);
           ^~~~~~~
1 warning generated.
```

Specify stack alignment to 4-bytes boundary

$ gcc -mpreferred-stack-boundary=2 FormatString.c -o FormatString
$ ./FormatString %s
This is a secret!

Solution: DO NOT depend on user supplied arguments directly

Heap overflow example

```
typedef struct chunk {
        char inp[64];
        void (*process)(char*);
}chunk_t;
void showlen(char *buf) {
        int len;
        len = strlen(buf);
        printf("%d", len);
}
int main(int argc, char**argv) {
        chunk_t *next;
        setbuf(stdin, NULL);
        next = malloc(sizeof(chunk_t));
        next->process = showlen;
        gets(next->inp);
        next->process(next->inp);
}
```

Global data overflow example

```
struct chunk {
        char inp[64];
        void (*process)(char*);
}chunk_t;
void showlen(char *buf) {
        int len;
        len = strlen(buf);
        printf("%d", len);
}
int main(int argc, char**argv) {
        setbuf(stdin, NULL);
        chunk.process = showlen;
        gets(chunk.inp);
        chunk.process(chunk.inp);
}
```

Defense Techniques

- Canary values
- Guard values
- Randomized instruction sets
- Address space layout randomization
- Padding

Canary values

- Randomizing the return address
  - Storing canary values next to the return address (if canary modified, attack happened)
    - Example: Random canaries
    - Example: XOR random canaries
- Integrity check for return address
  - Storing a copy of the return value in a safe place

Guard values

Used by Stack-Smashing Protector (ProPolice)

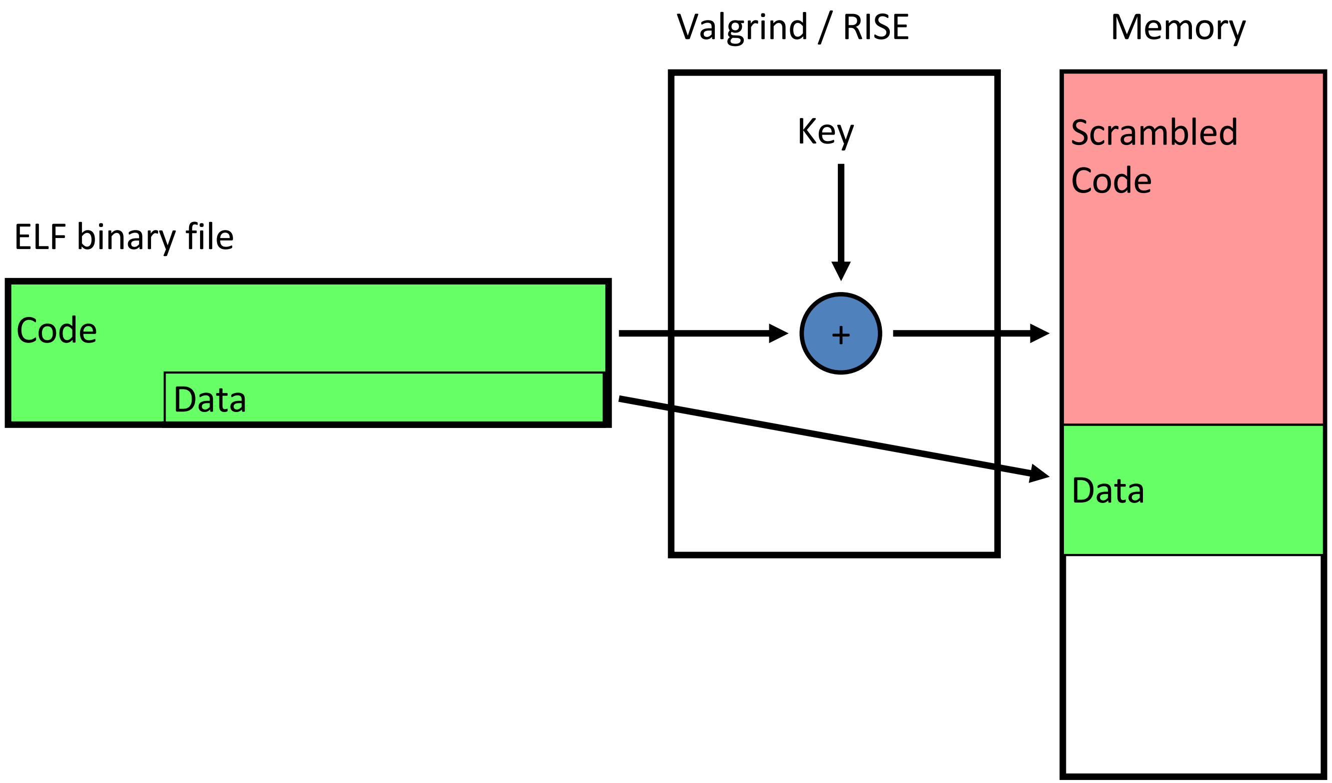
```
foo () {
   char *p;
   char buf[128];
   gets (buf);
}
```

```
Int32      random_number;
foo () {
   volatile int32 guard;
   char buf[128];
   char *p;
   guard = random_number;
   gets (buf);
   if (guard != random_number)
/* program halts */
}
```

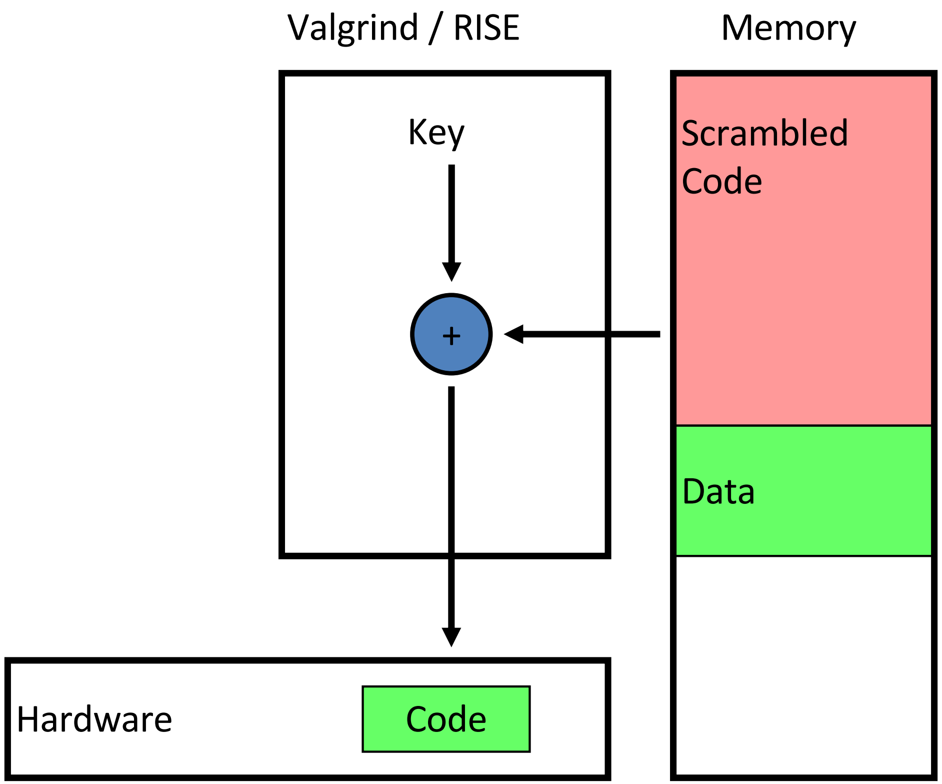
1. Insert guard instrument
2. Relocate local variables

-fstack-protector (when there is a byte array)
-fstack-protector-all (Always use guard instruments)

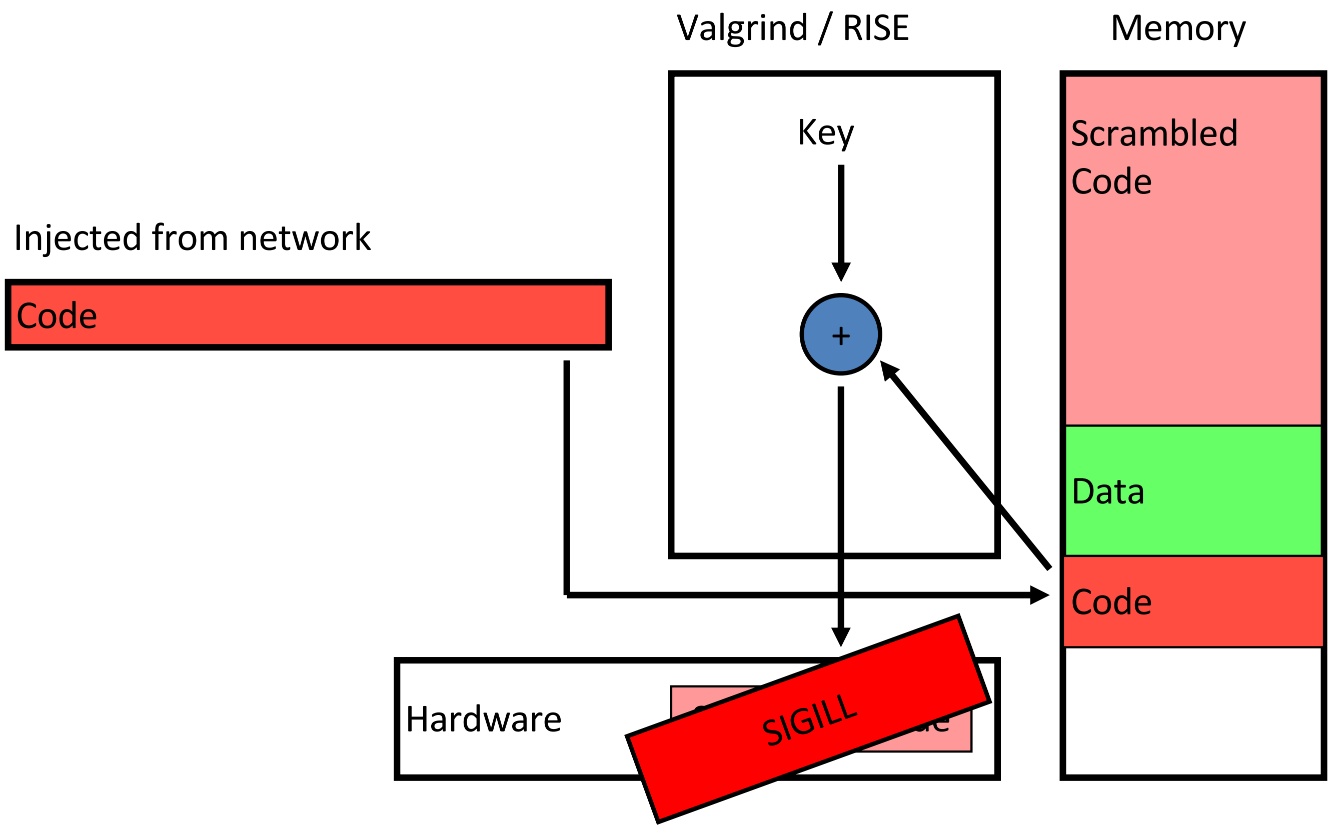
Randomizing instruction sets



Randomizing instruction sets



Randomizing instruction sets



Analysis of instruction sets rand.

- Pros
  - Excellent prevention against modified code
  - Usually, does not need modification of the original binary
- Cons
  - Performance penalties
  - Cannot prevent against data corruption
  - Needs extra protected resources, such as randomization algorithm, credentials, etc.

Stopping malware propagation

- Solution: Diversify software execution at runtime
- For example, diversify allocated memory addresses
- Expected result: the same copy of malware will stop when the second machine is attacked (due to different address allocation)
- Example systems:
  - Linux’s ASLR (Address Space Layout Randomization)

Address Space Layout Randomization

- Threat: memory error exploits
- Goal: remove predictability from memory access
- Example predictability
  - All stacks start at address x in an OS
- Solution:
  - Randomize absolute location of all code and data
  - Randomize relative distances b/t data items

Address Randomization methods

- **Randomize base addresses of memory regions**
  - Stack: subtract large value from stack pointer
  - Heap: allocate large blocks
  - Dynamic libraries: randomize base address (prevents return-to-libc)
  - Code/static data: re-link at different addresses

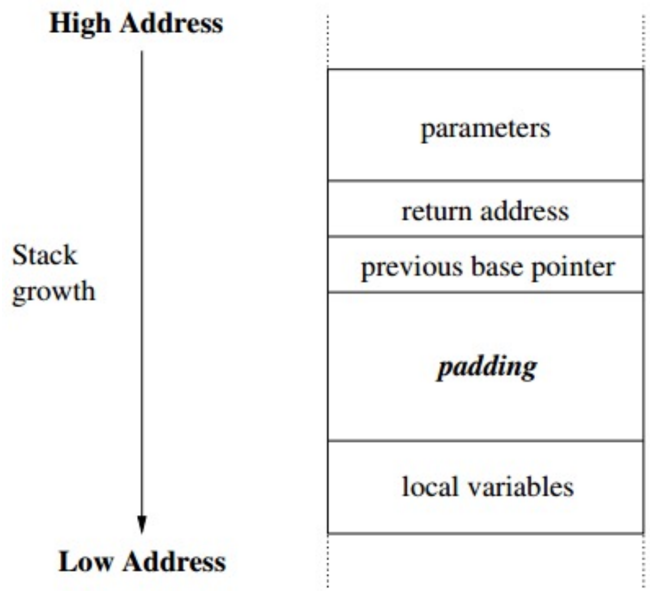
Address Randomization methods

- **Introduce dummy variables**
  - with randomized size and values to guard against predicting the location of the return address
- **Permute the order of variables/routines**
  - Local variables in stack frame
  - Order of static variables
    - Use random locations that are difficult to predict
  - Order of routines in dynamic libraries or executables

Address Randomization Problems

- Problems with rearrangement
  - Sometimes not possible to rearrange addresses
  - Example: local vars addresses of caller higher than that of callee
  - Example: Cannot rearrange malloc allocated memory
- Introduce random gaps between objects
  - Random padding into stack frames
  - Random padding between successive malloc requests
  - Random padding between variables in static area
  - Introduce gaps between routines with added jump instructions

Example padding approach



Analysis of ASLR

- Pros
  - Slows down stack smashing attacks
  - Disallows absolute address-based attacks
- Cons
  - Limited to the randomization algorithm
  - Address space size is limited, brute force possible
  - All memory regions, no exception, must be randomized