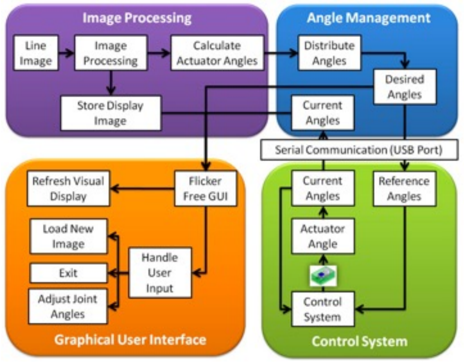
Arms Race in Memory Error Exploit and Defense

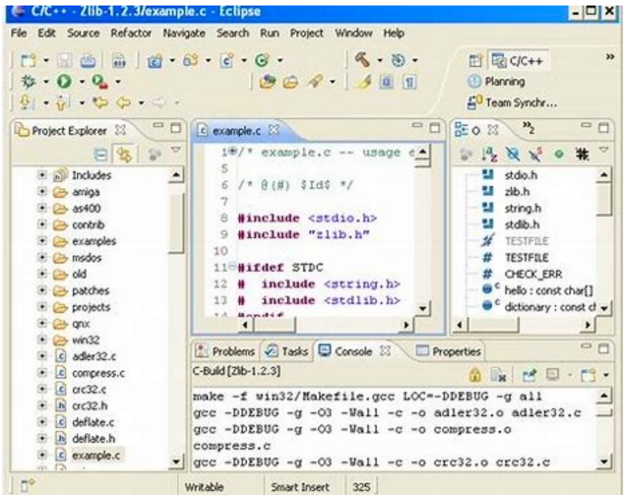
Liang Zhenkai 梁振凯

NUS National University of Singapore

School of Computing

Life Cycle of Software

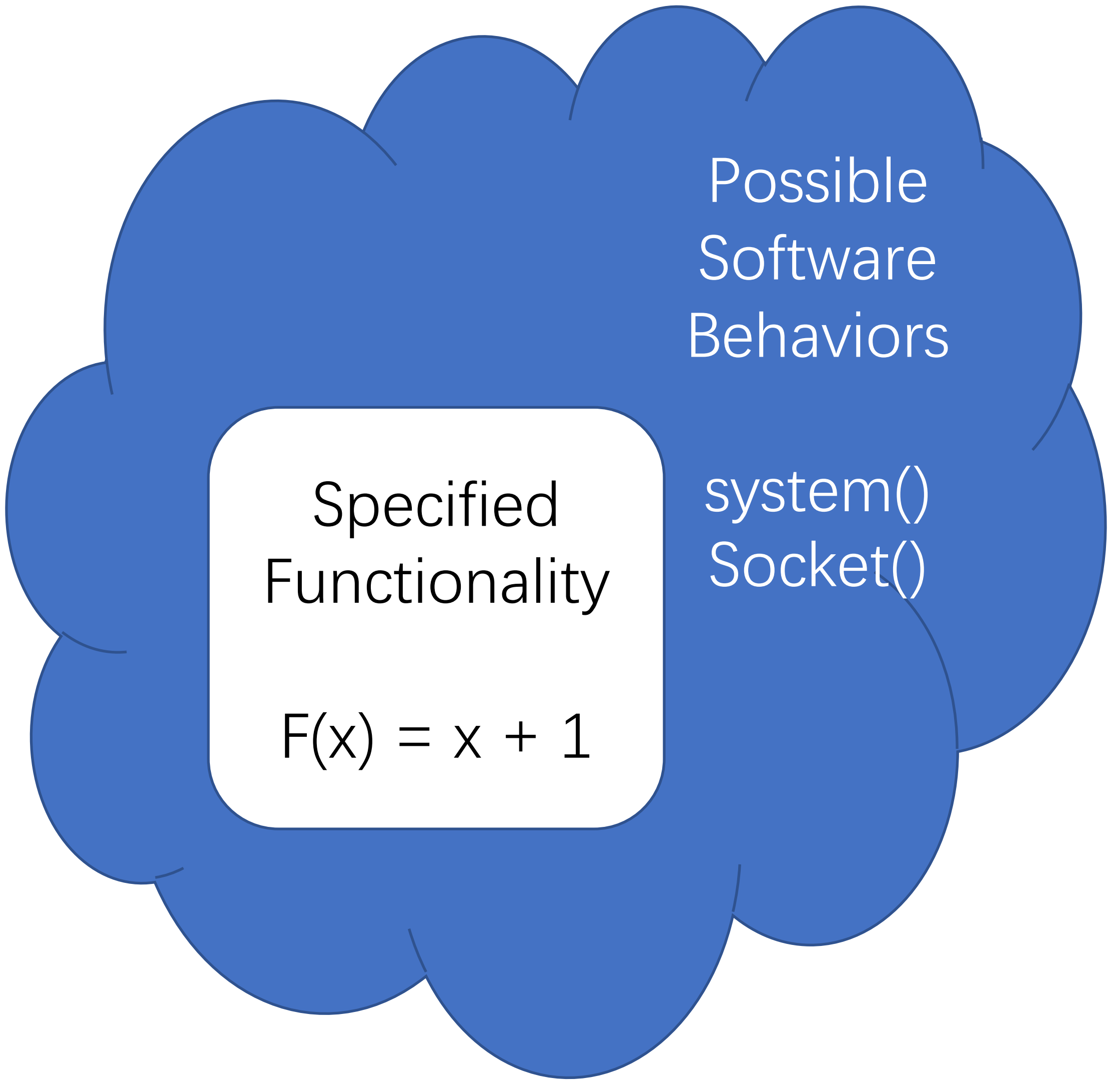




- Losses and gains in software development
  - Information discarded for efficiency: type, structure, ⋯
  - Additional functionality: library, compiler addition, bug, ⋯

Functionality, Flexibility, and Security

- Security is about "nothing else"
  - Specified functionality and **only** specified functionality
- Flexibility is the root of many security problems

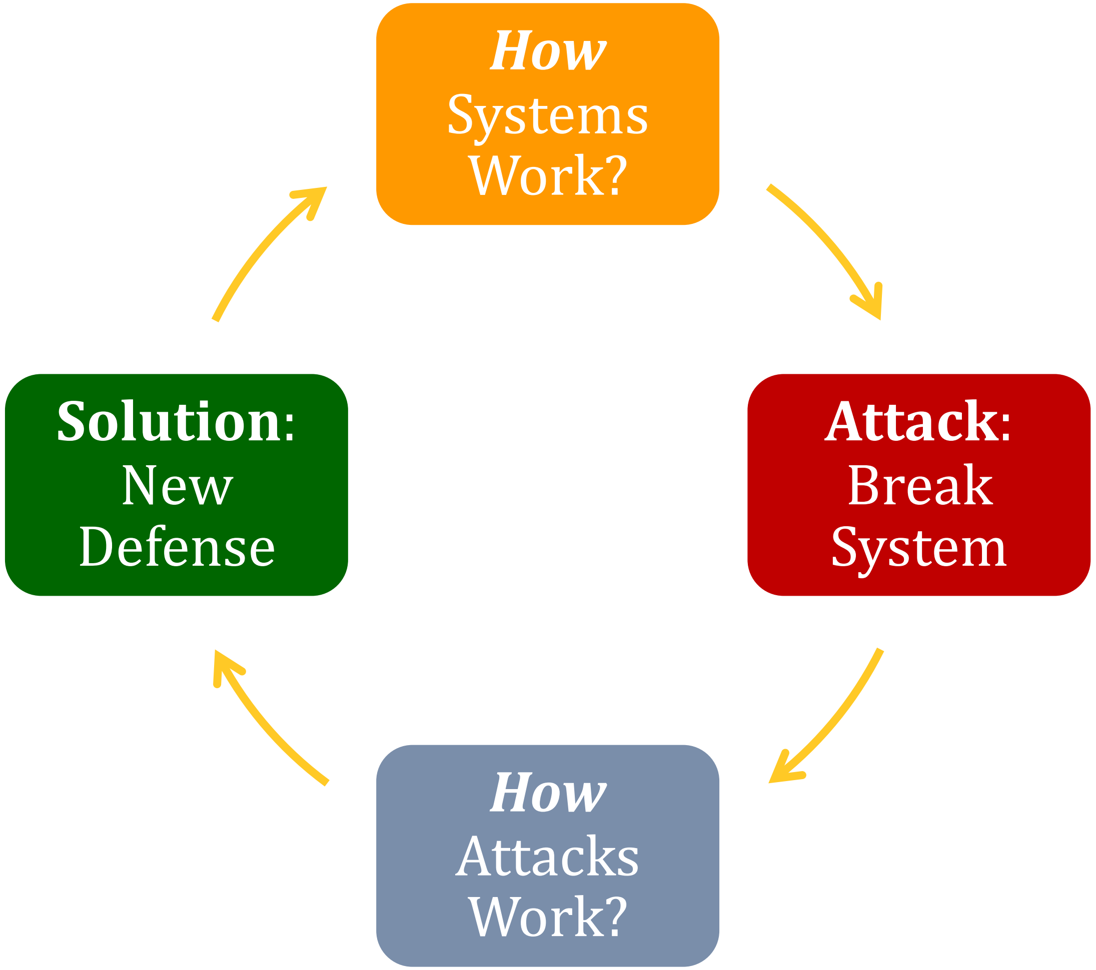


Simplicity in System Design

- KISS
  (Keep It Simple, Stupid)

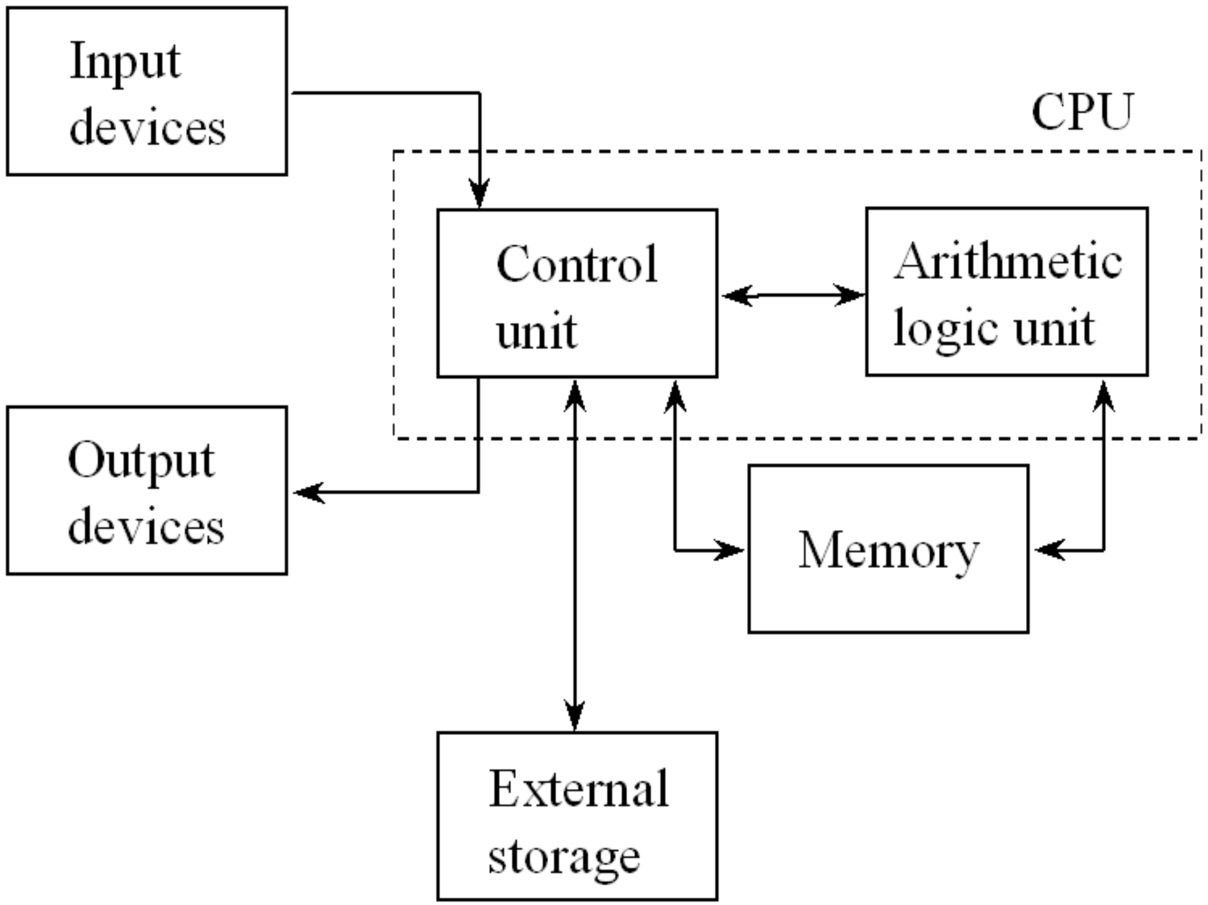
- KICS
  (Keep It Complex & Smart)

Arms Race between Attackers and Defenders



Basis of Function Call Mechanism

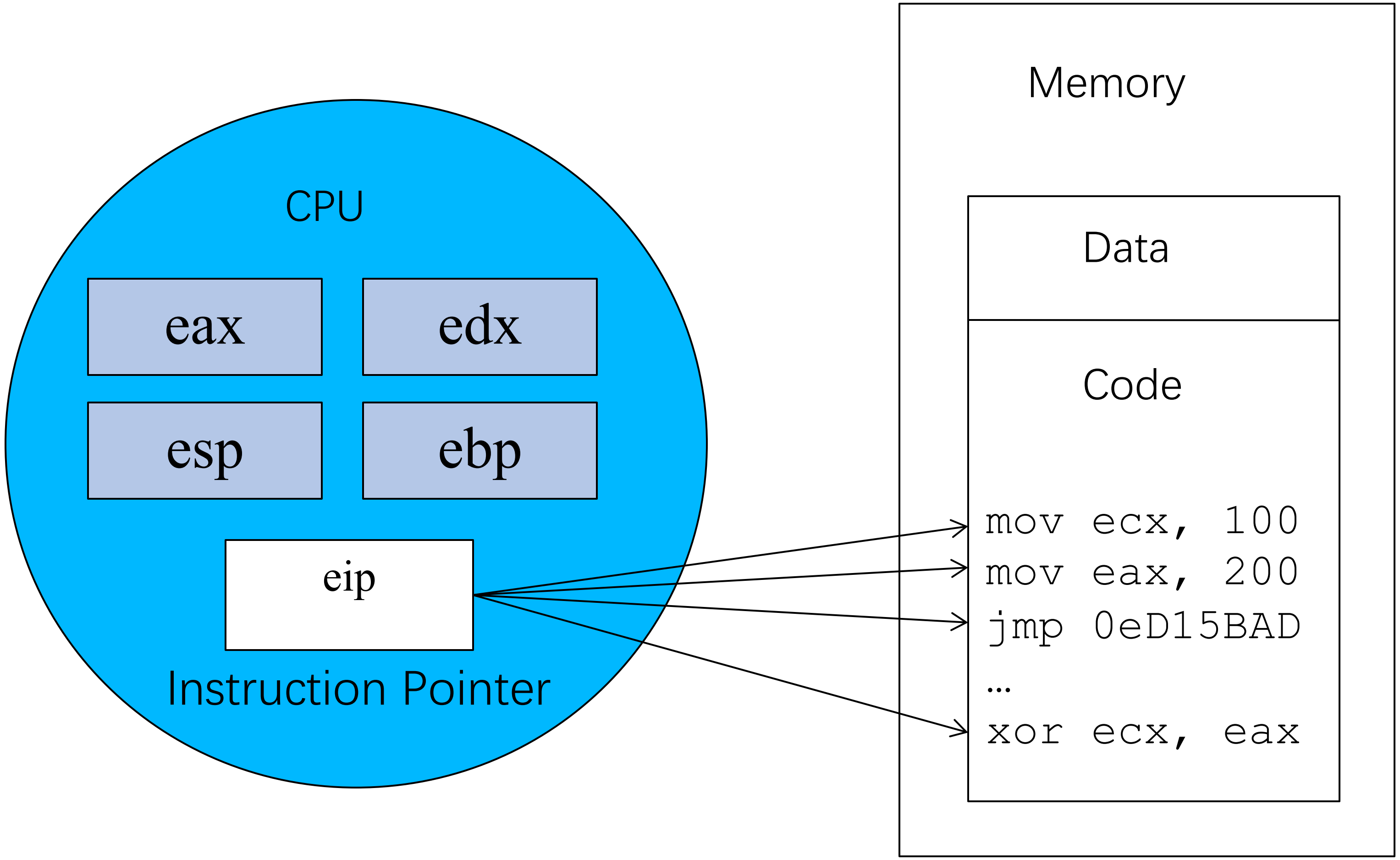
von Neumann Architecture



Implication to Computer Security

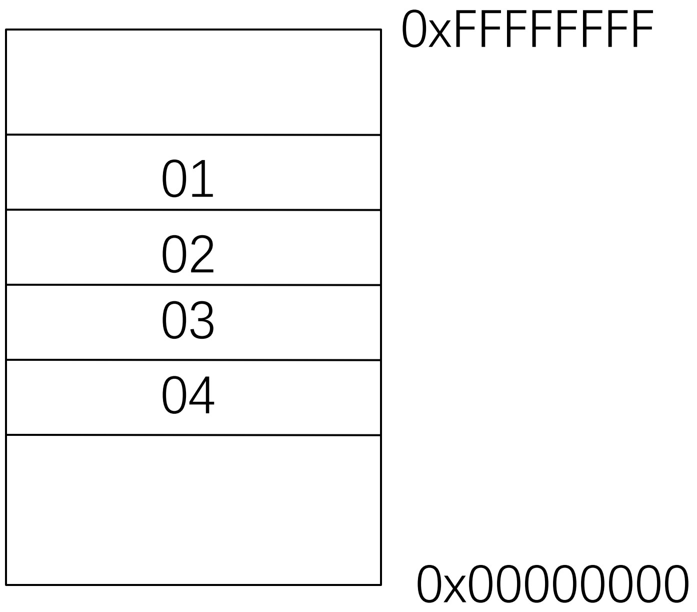
- What is special about von Neumann architecture?
  - What is its connection to vulnerabilities & malware?
- Keromytis A. D., "*von Neumann and the Current Computer Security Landscape*", 2008:
  - Code is treated as data
  - Programs may be tricked into treating input data as code: basis for all code-injection attacks!

CPU and Memory



Program Representation in Memory

- Both code and data are represented as numbers
- Code:
  - `lea ecx, [esp+4]` represented as `0x8d 0x4c 0x24 0x04`
- Data:
  - On Intel CPUs, least significant bytes is put at lower addresses
  - It is called little endian
  - For example, `0x01020304`



Function Calls

- Functions break code into smaller pieces
  - Facilitating modular design and code reuse
- A function can be called in many program locations
- How does it know where it should continue after it finishes?

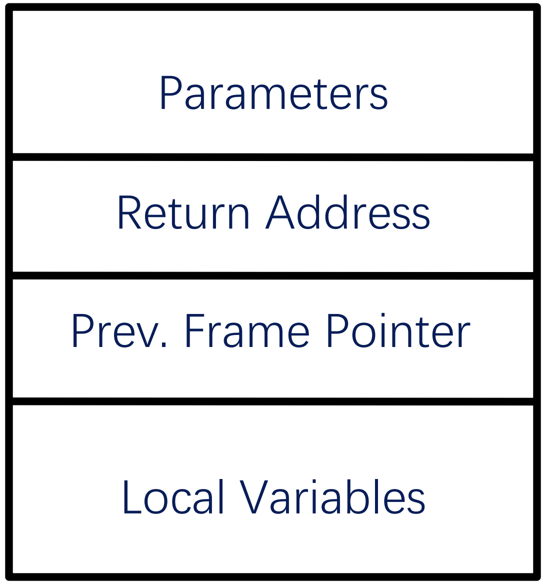
```
void sample_function(void)
  {
              char buffer[10];
              printf("Hello!\n");
              return;
  }
main()
  {
              sample_function();
              printf("Loc 1\n");
              sample_function();
              printf("Loc 2\n");
  }
```

Stack

- A data structure storing important information for each process running on a computer
- Last in, first out (LIFO)
- Stack operations:
  - push
  - pop
  - top
- In Intel systems, stack grows from high address to low address

Activation Record

- Each call of a function has an activation record (stack frame):
  - Parameters
  - Return address
  - Previous frame pointer
  - Local variables

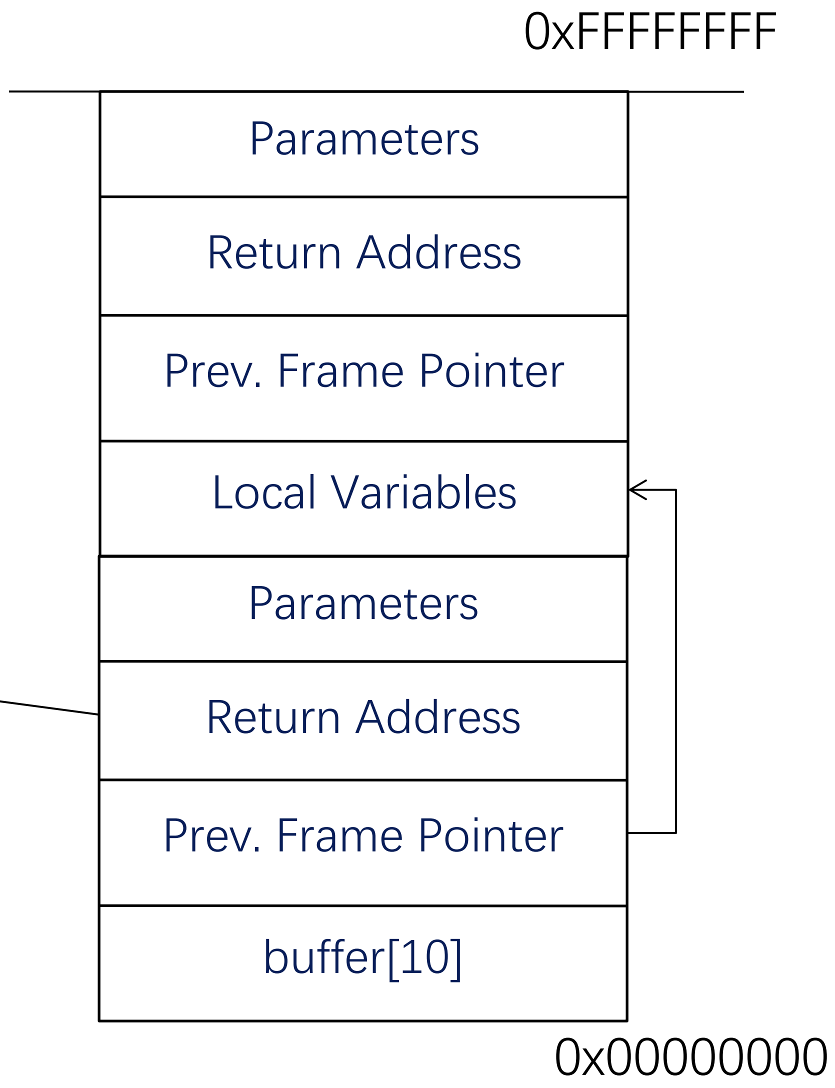


Steps of Call and Return

- Caller:
  - Save registers
  - Push parameters on stack
  - Push return address on stack
  - Jump to the beginning of function
- Callee:
  - (Optional) Save frame pointer (EBP), and set frame pointer to stack top
  - Allocate local variables
- Callee:
  - Set return values
  - Deallocate local variables
  - (Optional) restore frame pointer
  - Jump to the return address on stack
- Caller:
  - Get return values
  - Pop parameters from stack
  - Restore saved registers

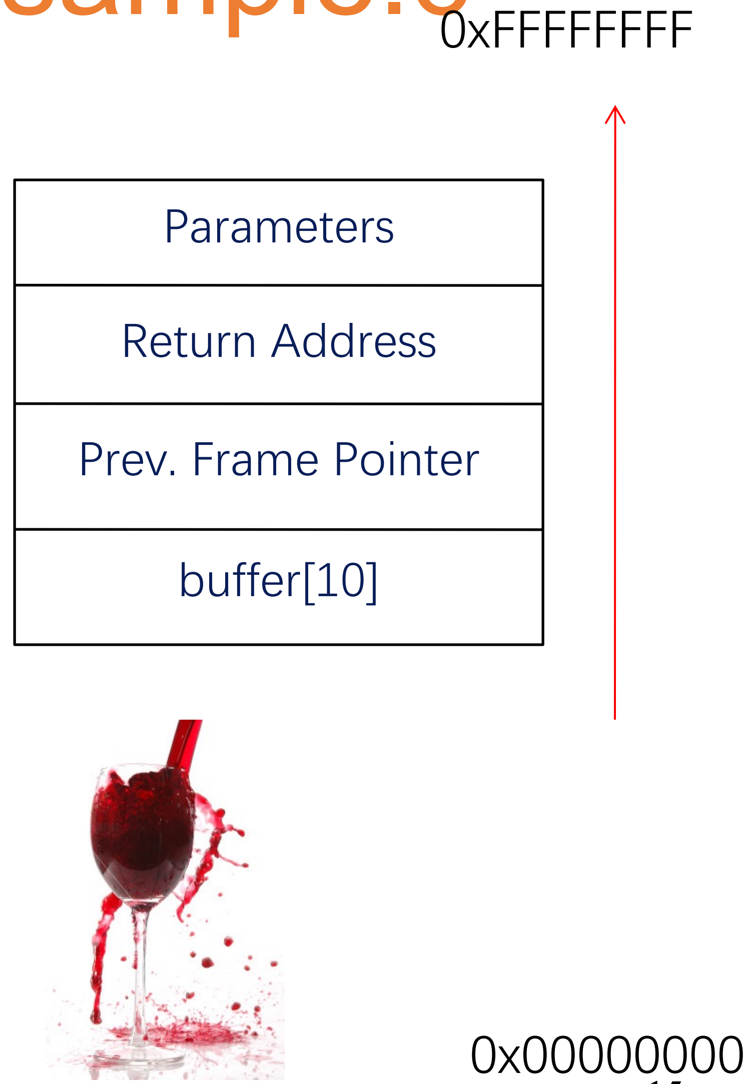
Stack Action Illustrated

```
void sample_function(void)
  {
     char buffer[10];

     printf("Hello!\n");
     return;
  }

main()
  {

     sample_function();
     printf("Loc 1\n");

     sample_function();
     printf("Loc 2\n");
  }
```



Observation on sample.c

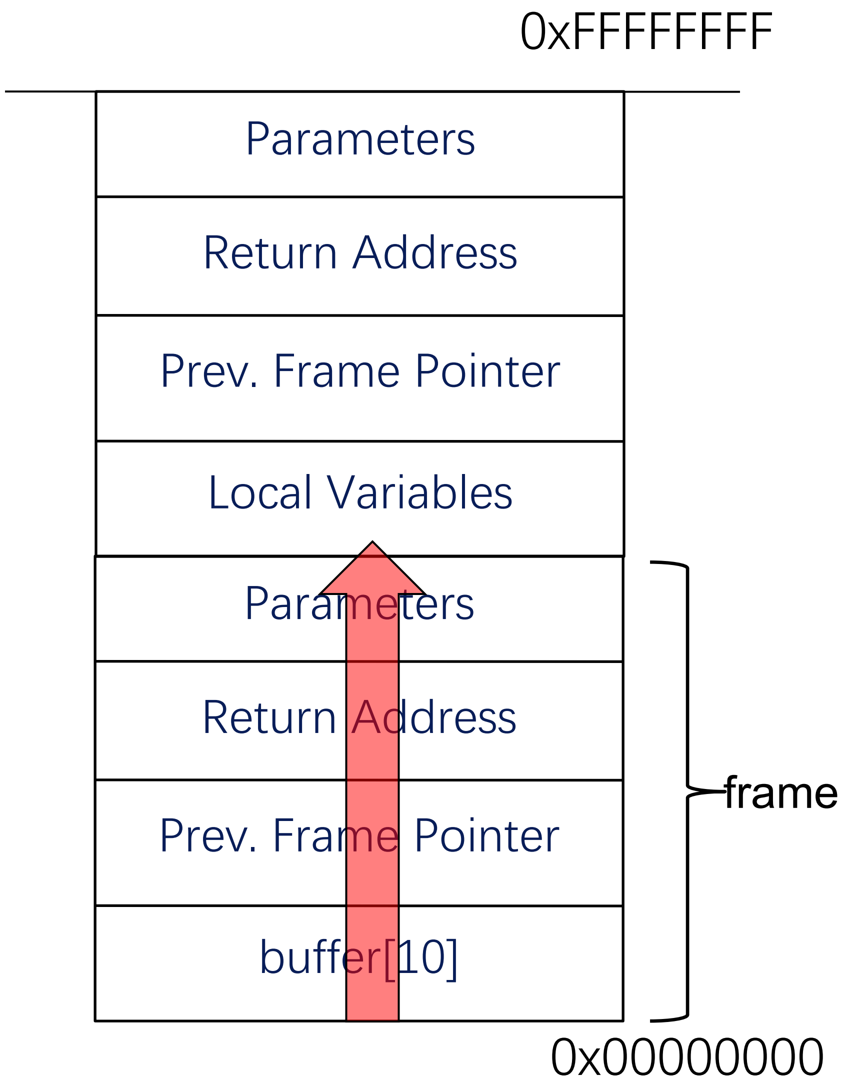
- Buffer grows toward return address
- If we more than 10 bytes for array buffer, the content will spill into adjacent memory region, previous frame pointer, then return address



Buffer Overflow Attacks

Stack-Smashing Buffer Overflow

```
void sample_function(void)
  {
        char buffer[10];
        gets(buffer);
        return;
  }

main()
  {

        sample_function();
        printf("Loc 1\n");

        sample_function();
        printf("Loc 2\n");
  }
```

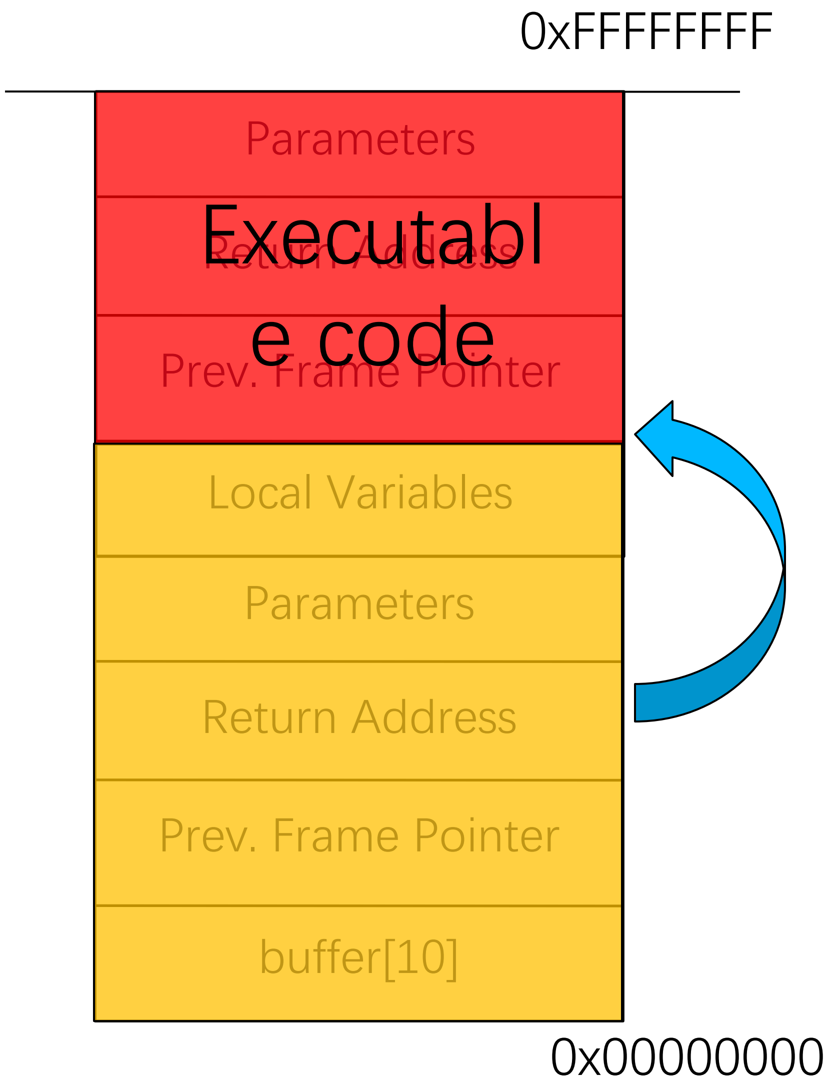


Result of Buffer Overflow Example

- Return address is overwritten by user inputs:
  - Program will "return" to the new address after finishing the function with vulnerable buffer
- If the overwritten return address is an invalid memory address, program will crash
  - What if its not invalid?
- Where is attacker's malicious code?

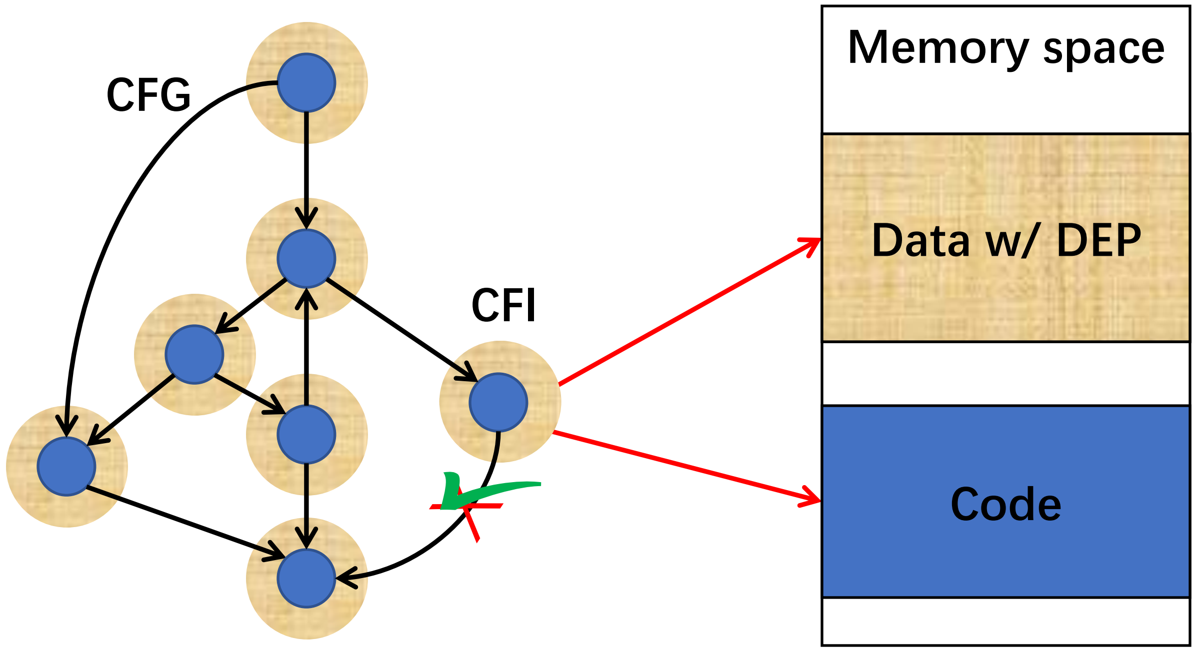
Malicious Code Injection

- Remember executable code is also represented as bytes (von Neumann architecture)
- Attackers can include code in the input
  - Called **shell code**
- They can arrange the return address to point to the injected code



Control Attacks and Defenses

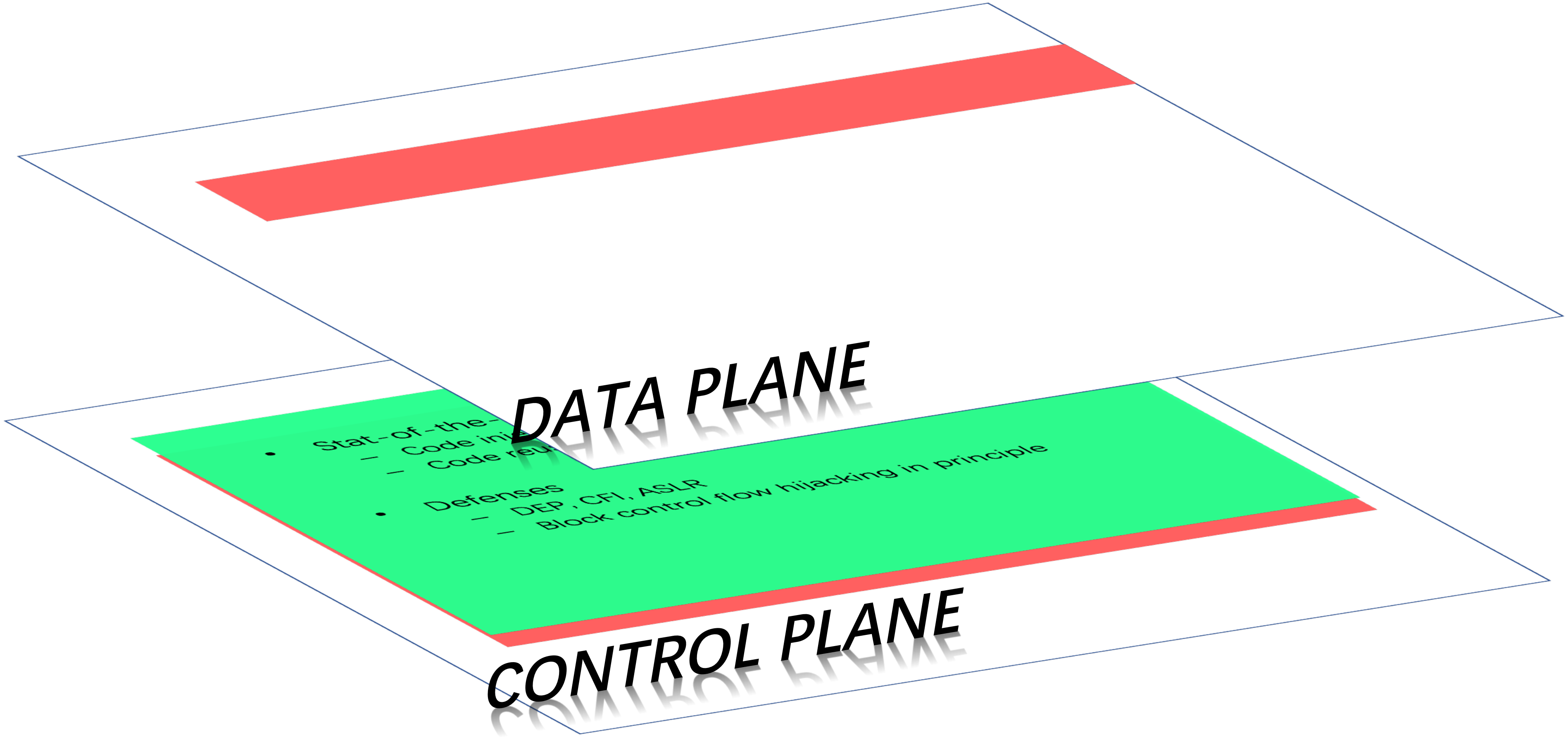
- Code injection ⟵ Data Execution Prevention
- Code reuse ⟵ Control Flow Integrity
  - return-to-libc
  - return-oriented programming (ROP)



Arms Race in Memory Space

- State-of-the-art exploits
  - Code injection
    - Buffer overflow/heap spray
  - Code reuse
    - Ret2libc, ROP
    - Control-flow bending

- Defenses
  - Data Execution Prevention
  - Control Flow Integrity


DATA PLANE
Defenses
DEP, CFI, ASLR
Block control flow hijacking in principle
CONTROL PLANE

Data-Oriented Exploits

- Corrupting non-control data
  - Legitimate control flow
  - Significant damage

```
// set root privilege
seteuid(0);
......
// set normal user privilege
seteuid(pw->pw_uid);
// execute user's command
```

Wu-ftpd *setuid* operation*

```
//0x1D4, 0x1E4 or 0x1F4 in JScript 9,
//0x188 or 0x184 in JScript 5.8,
safemode = *(DWORD *)(jsobj + 0x188);
if( safemode & 0xB == 0 ) {
      Turn_on_God_Mode();
}
```

IE *SafeMode* Bypass+

* Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In USENIX 2005.

+ Yang Yu. Write Once, Pwn Anywhere. In Black Hat USA 2014

Data-Oriented Programming

Non-Control Data Attacks

- Corrupt/leak several bytes of security-critical data

```
//set root privilege   *
seteuid(0);
......
//set normal user privilege
seteuid(pw->pw_uid);
//execute user's command
```

```
//offset depends on IE version +
safemode = *(DWORD *)
            (jsobj + offset);
if(safemode & 0xB == 0) {
    Turn_on_God_Mode();
}
```

- Special cases relying on particular data/functions
  - user id, safemode, private key, etc
  - interpreter – printf(), etc

specific

trivial-to-prevent

- *What is the expressiveness of general non-control data attacks?*

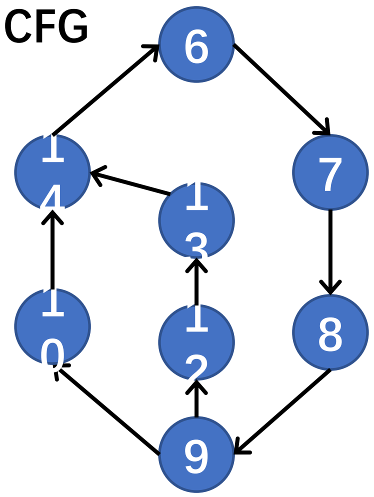
* Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In USENIX 2005.
\+ Yang Yu. Write Once, Pwn Anywhere. In Black Hat USA 2014

Motivating Example

```c
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
  readData(sockfd, buf);        // stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  } //...(following code skipped)...
}
```

Vulnerable Program

CFG



?

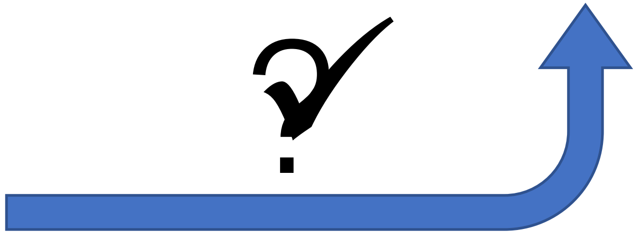
```c
struct Obj {struct Obj *next; int prop;}

void updateList(struct Obj *list, int addend){
 for(; list != NULL; list = list->next)
   list->prop += addend;
}
```

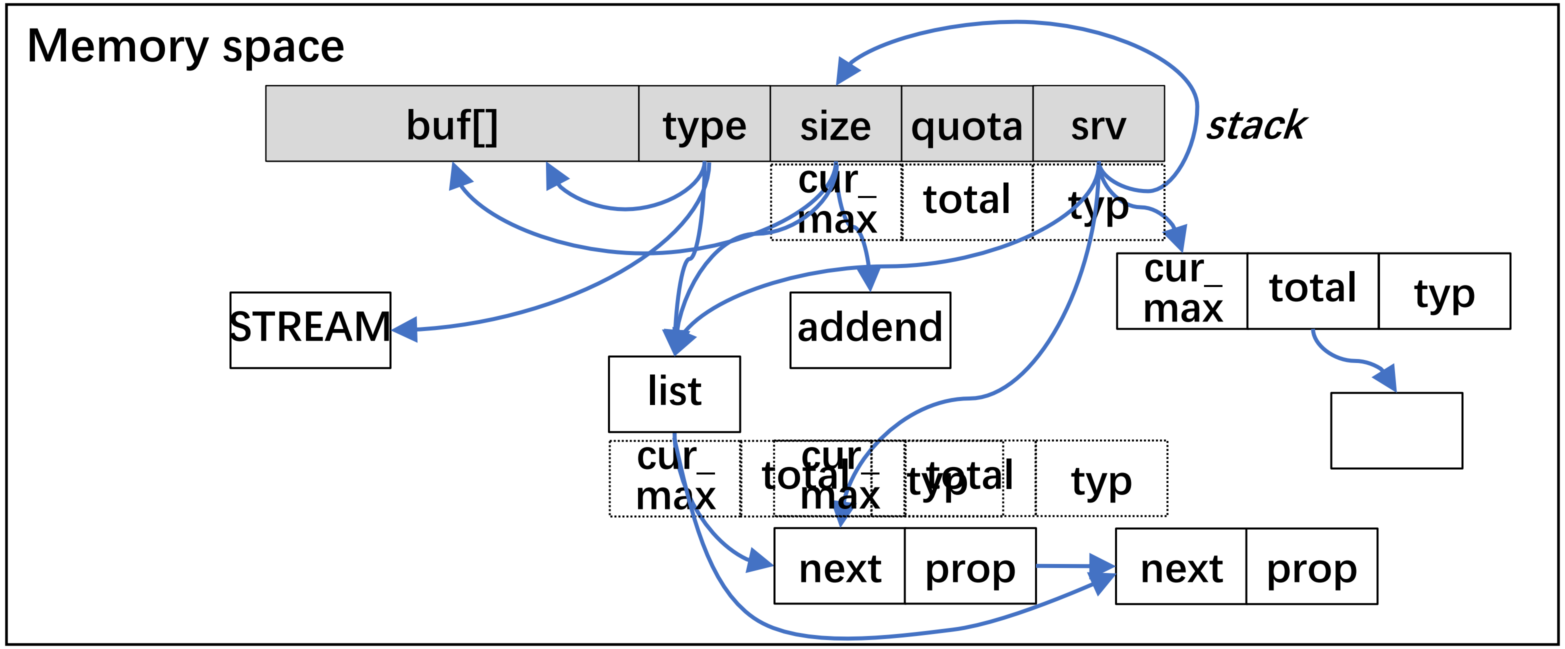
Expected Computation

Motivating Example (cont.)

```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

```
for(; list != NULL; list = list->next)
  list->prop += addend;
```





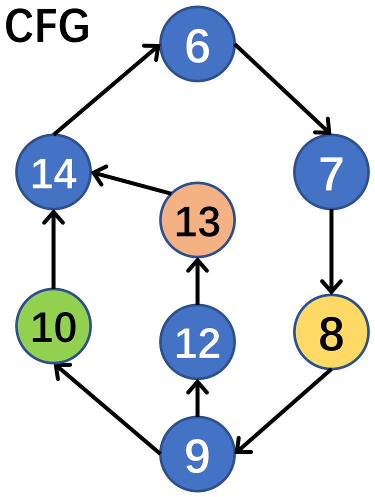
Data-Oriented Programming (DOP)

- General construction
  - w/o dependency on security-critical data / functions
- Expressive attacks
  - towards Turing-complete computation
- Rely on data-oriented gadgets & dispatchers

```
while (quota--) {
  readData(sockfd, buf);     //stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  } //...(following code skipped)...
}
```

Data-Oriented Gadgets

- x86 instruction sequence
  - Shown in normal execution
  - Simulating registers with memory
  - ***Load*** *micro-op --> **Semantics** micro-op --> **Store** micro-op*

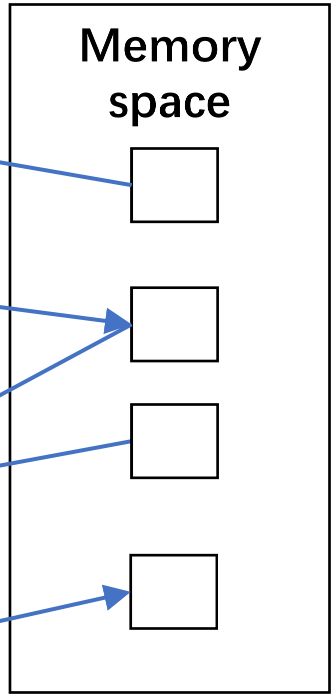


Addition: `srv->total += *size;`

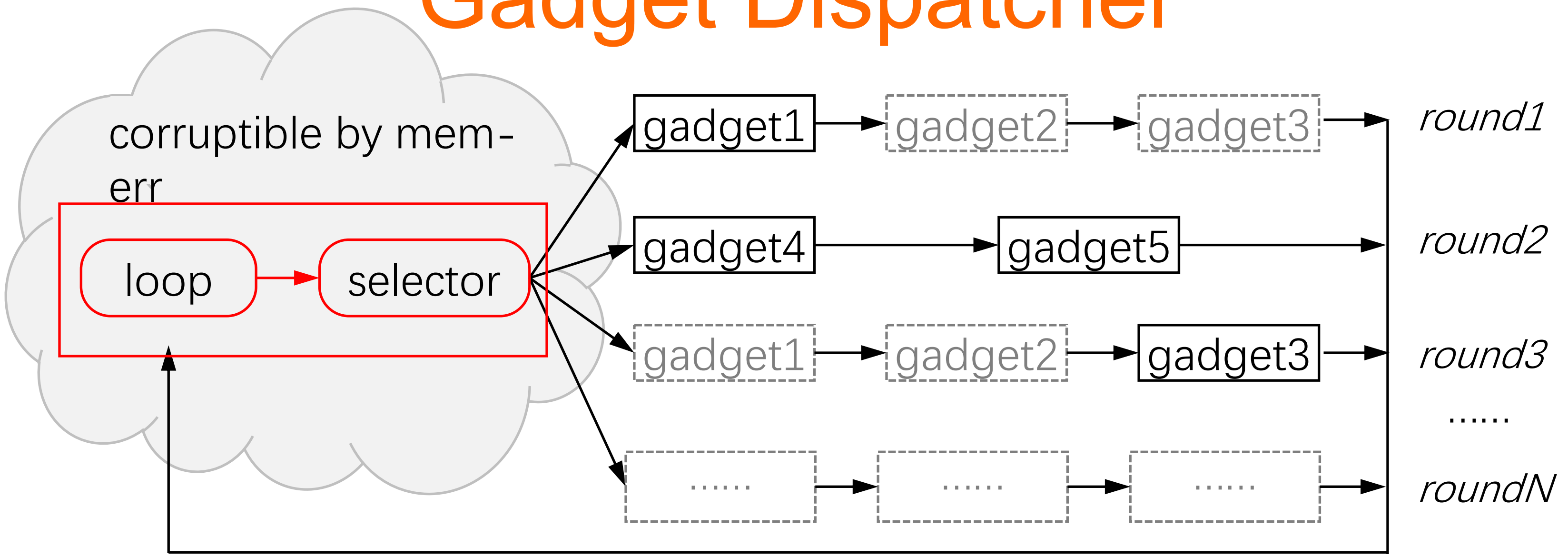
```
mov (%esi), %ebx     //load micro-op
mov (%edi), %eax     //load micro-op
add %ebx, %eax       //addition
mov %eax, (%edi)     //store micro-op
```

Load: `*size = *(srv ->cur_max);`

```
mov (%esi), %ebx     //load micro-op
mov (%edi), %eax     //load micro-op
mov 0xb(%ebx), %eax  //load
mov %eax, (%edx)     //store micro-op
```



Gadget Dispatcher



- Chaining data-oriented gadgets
  - **Loop** ---> repeatedly invoke gadgets
  - **Selector** ---> selectively active gadgets

```
while (quota--) {                         // loop
  readData(sockfd, buf);                  // selector
  if(*type == NONE ) break;
  if(*type == STREAM) *size = *(srv->cur_max);
   else{ srv->typ = *type;  srv->total += *size; }
}
```

Turing Completeness

- DOP emulates a minimal language *MINDOP*
  - *MINDOP* is Turing-complete

Semantics	Statements In C	Data-Oriented Gadgets in DOP
arithmetic / logical	a op b	*p op *q
assignment	a = b	*p = *q
load	a = *b	*p = **q
store	*a = b	**p = *q
jump	goto L	vpc = &input
conditional jump	if (a) goto L	vpc = &input if *p
p – &a;	q – &b;	op – any arithmetic / logical operation

Attack Construction

```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
   } //...(code skipped)...
}
```

- Gadget identification
  - statically identify load-semantics-store chain from LLVM IR
- Dispatcher identification
  - static identify loops with gadgets from LLVM IR
- Gadget stitching
  - select gadgets and dispatchers (manual)
  - check stitchability (manual)