Memory Corruption 2

Advanced Internet Security

Adrian Dabrowski, **Georg Merzdovnik**, Aljosha Judmayer, Johanna Ullrich, Markus Kammerstätter, Stefan Riegler, Christian Kudera

News from the Lab

- 75 solved Challenge 1
- Challenge 2 starts tomorrow

News from the Field

- The DUHK Attack[1] [2]
  - Don't Use Hard-coded Keys
  - Allows to decrypt encrypted traffic
- Attack against old Fortinet Products (still in use)
  - Base on vulnerability in pseudorandom number generator together with hard coded seed
    - X9.31 RNG was removed from The Federal Information Processing Standard FIPS list of approved RNGs in 2016
    - U.S. government computer security standards for requirements in cryptographic modules
    - However the basic attack was described in 1998 already, but the Standard was not updated
    -
- So this is actually less about the attack itself but on how to handle certifications in general

[1] https://duhkattack.com/
[2] https://blog.cryptographyengineering.com/2017/10/23/attack-of-the-week-duhk/

News from the Field - IoT Botnets all over again

- ”Reaper” or ”IoTroop”
  - Security company announced they are tracking new Botnet ”forming to create a cyber-storm that could take down the Internet”[3]
  - Quotes from their blog entry…
    - A massive Botnet is forming to create a cyber-storm that could take down the internet.
    - An estimated million organizations have already been infected.
  - More advanced than Mirai
    - Does **not only** use factory-default or hard-coded usernames and passwords
    - Also exploits other known (unpatched) vulnerabilities…
    - One from this year (CVE-2017-8225) but also another one in Linksys routers known since 2014

- We still lack a good process for software updates on these devices...

[3] https://research.checkpoint.com/new-iot-botnet-storm-coming/

Format String Exploitation

A short introduction to printf

- *printf()
  - function with variable number of arguments `int printf(const char *format, ...)`
  - as usual, arguments are fetched from the stack
- `const char *format` is called format string
  - used to specify type of arguments
  - %d or %x for numbers
  - %s for strings

Format String Example


```c
#include <stdio.h>

int main(int argc, char **argv){
    char buf[128];
    int x = 1;

    snprintf(buf, sizeof(buf), argv[1]);
    buf[sizeof(buf) - 1] = '\0';

    printf("buffer (%d): %s\n", strlen(buf), buf);
    printf("x is %d/%#x (@ %p)\n", x, x, &x);
    return 0;
}
```

What is the Problem?

- User input passed as format string
  - Allows user to pass format string which will be interpreted

```
printf("Hello world\n"); // is ok
printf(user_input);      // vulnerable
```

- Allows to read values with format identifiers

Simple Example

- `prinf("%x%x%x%x")`

```
"AAAABBBB-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p"

    AAAABBBB-0x7fffc0d0e030-(nil)-0x400-0x7f2cc0834400-
    0x7fffc0d0e188-0x200000000-0x4242424241414141-
    0x252d70252d70252d-0x2d70252d70252d70-0x70252d70252d7025-
    0x252d70252d70252d
```

- If you look closely, you will notice that our format string is also on the stack

Leaking Data

- This means we can read data from the stack
- But the approach is somehow limited
  - Possibly limited input (format string) length
- So, can we improve this to read arbitrary data?

$ Modifier

- actually we can give an **index** to format specifiers
  - Direct Parameter access
- "%<n>$p"
  - Tells the interpreter to take the n^{th} argument from the stack

```
"AAAABBBB%5$p"
     AAAABBBB0x7ffc0ccbfdb8

"AAAABBBB%7$p"
     AAAABBBB0x4242424241414141
```

Writing Arbitrary Data?

- ok, we can read anything
- but, can we also write something

Wrtiting Arbitrary Data!

- **%n**
  - from *man 3 printf*
    - *The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. (This specifier is not supported by the bionic C library[4].)*
  - This means we can write to the stack
- We can use the width modifier to write arbitrary values
  - for example, %.500d
  - even in case of truncation, the characters that would have been written are used for %n

[4] Google's standard C library for Android

Taking Control of the Program

- Overwrite function pointer
- e.g. GOT entries

The Heap

Heap Management

- Implementations

Algorithm	Operating System
dlmalloc	Doug Lea' malloc (general purpose)
ptmalloc2	GNU LibC (based on dlmalloc)
jemalloc	FreeBSD and Firefox
tcmalloc	Google (thread-caching malloc)
….	….

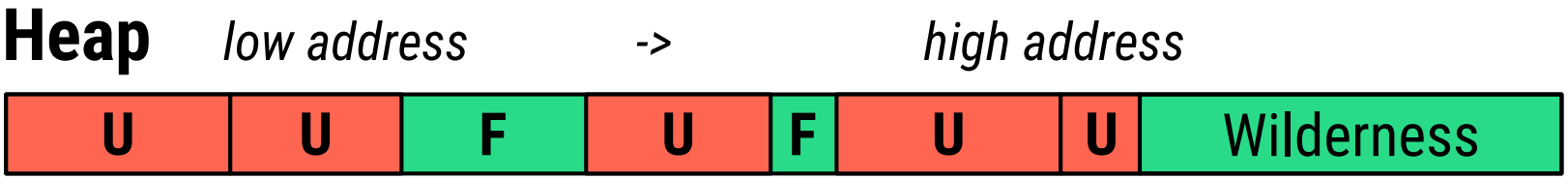
- Each application can use/implement it's own allocator

GlibC Memory Allocation

- Glibc integrated ptmalloc2
  - there may be differences now between these two
- Previously dlmalloc, but ptmalloc allows for better handling of threads
  - No need for locking/synchronisation
  - **Per-thread arena**

Glibc Malloc

- Memory Layout
  - heap is divided into continuous chunks of memory




```
U           ... used chunk
F           ... free chunk
Wilderness  ... topmost free chunk
```

- Wilderness chunk
  - only chunk that may be increased (with system call sbrk)
  - treated as bigger than all other chunks
    - If nothing else fits it will just be increased

Glibc Malloc

- Memory Chunk
  - continuous region of heap memory
  - can be allocated, freed, split, joined (two free chunks)
- Public and Internal routines

```
malloc(size_t n)
calloc(size_t unit, size_t quantity)
realloc(void* ptr, size_t n)
free(void *ptr)
```

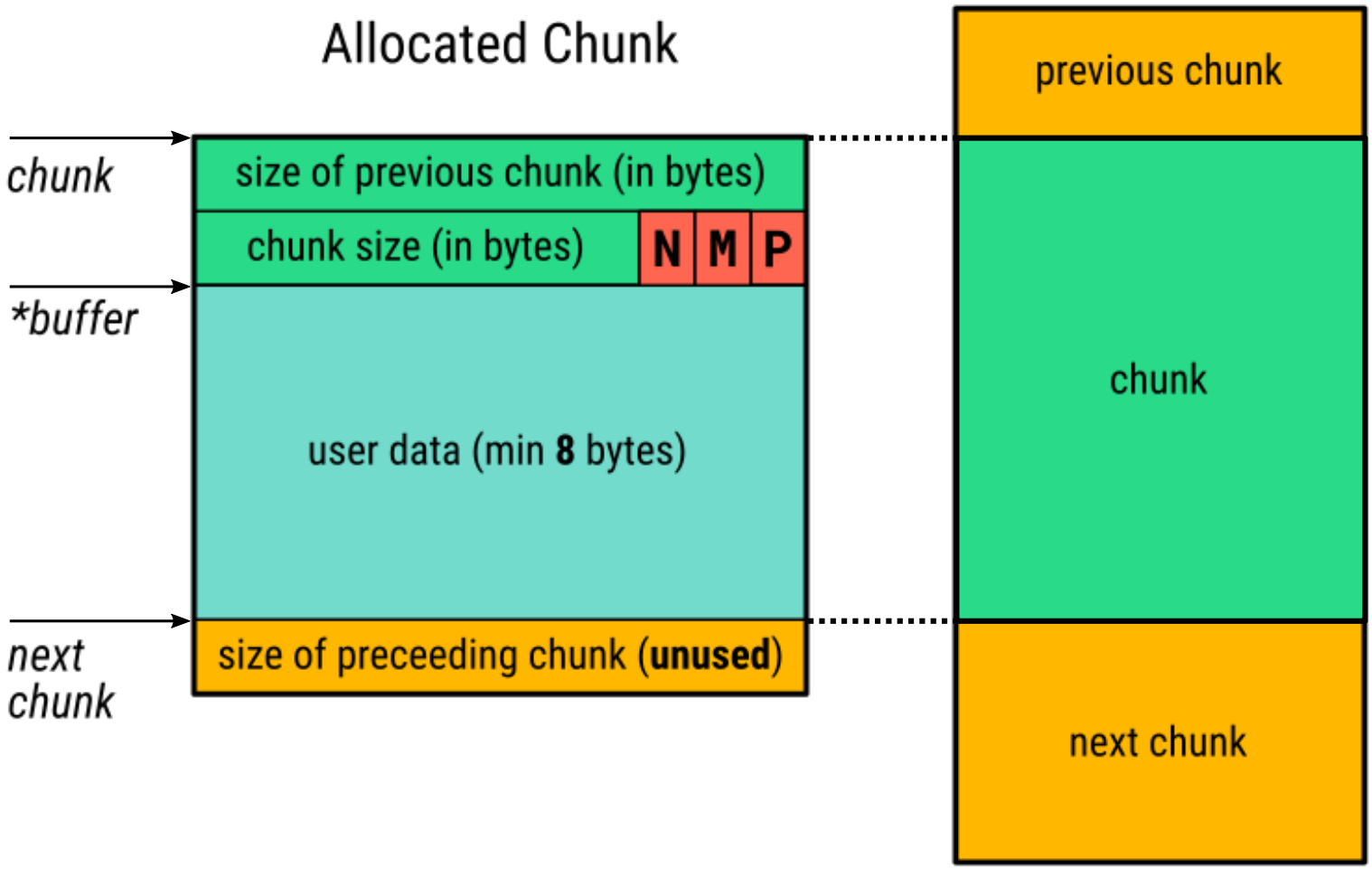
Glibc Malloc

- Boundary tag
  - holds chunk management information
  - stored in front of each chunk
  - 16 bytes large -> minimum allocated size “‘ struct malloc_chunk {

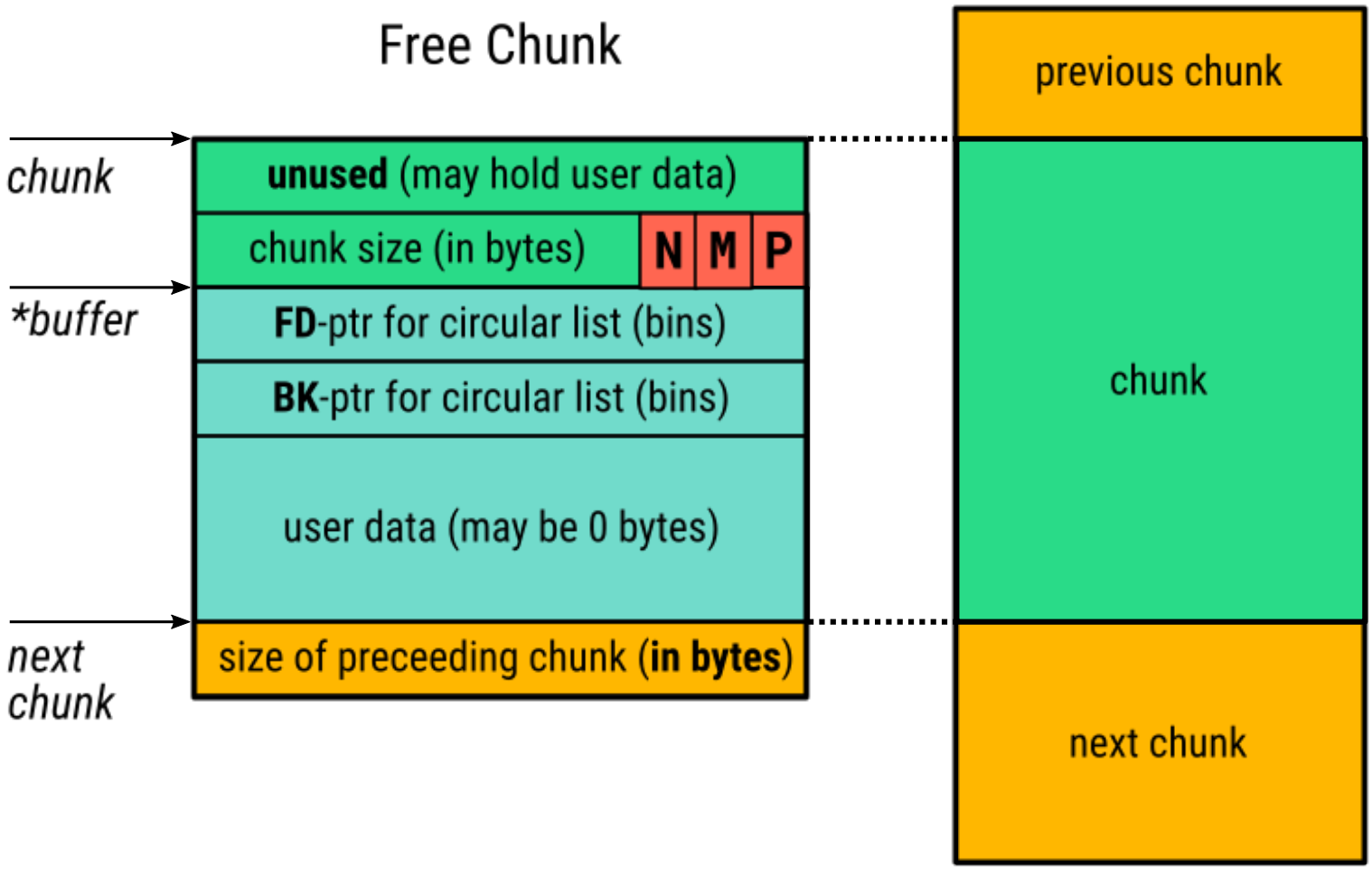
```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead.

  struct malloc_chunk* fd;         /* double links -- used only if free.
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free.
  struct malloc_chunk* bk_nextsize;
};
```

- pointer returned by malloc (for user) starts at fd
  - usually 8 bytes overhead for allocated chunks

Glibc Malloc



Glibc Malloc



Glibc Malloc

- Status Bits
  - Lower 3 bits of chunk size
  - Chunk size is always 8-byte aligned, so these would be unused otherwise

```
0x01 PREV_INUSE // set when previous chunk is in use
0x02 IS_MMAPPED // set if chunk was obtained with mmap()
0x04 NON_MAIN_ARENA // set if chunk belongs to a thread arena
```

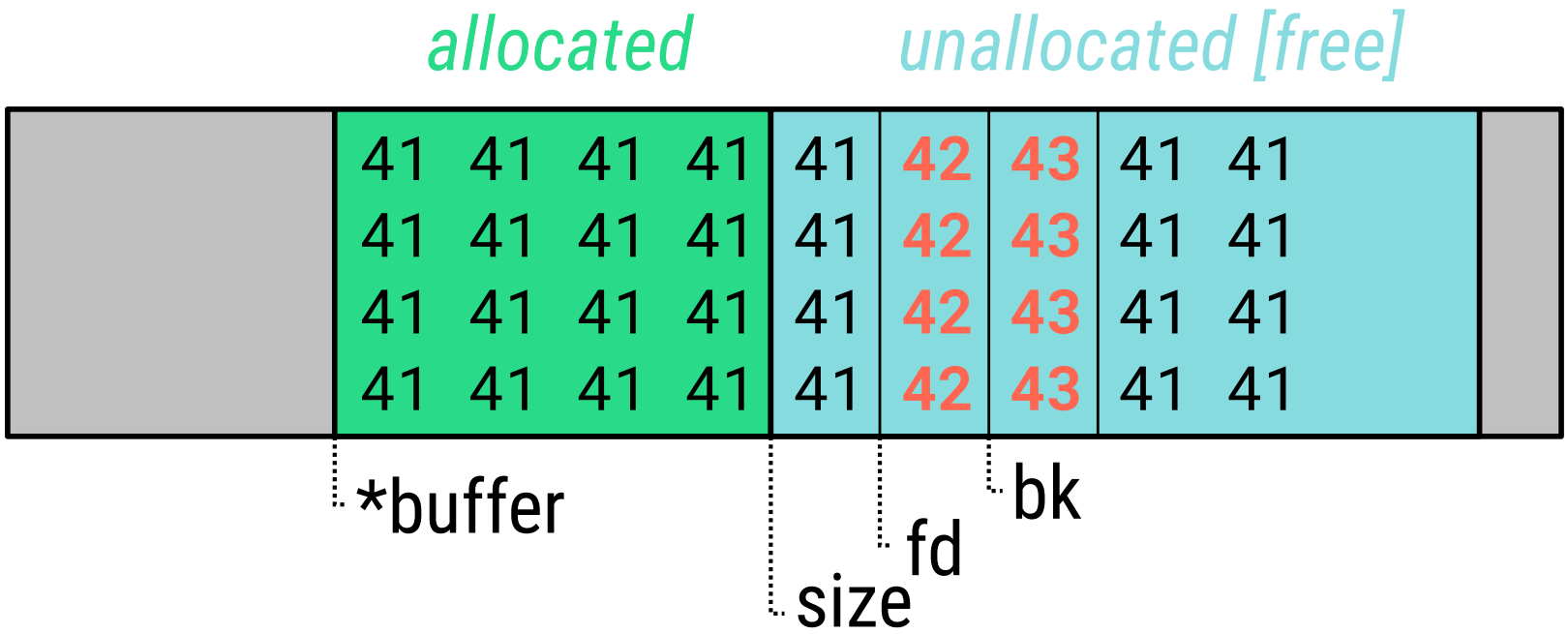
Glibc Malloc

- Bin Management
  - available chunks are stored in bins on a circular doubly-linked list
  - each bin holds chunks of a certain size range
  - the bin itself consists of two pointers (forward/back) and acts as the corresponding list head
  - each bin is initially empty
  - chunks are maintained in decreasing sorted order by size
    - best fit algorithm

Heap Overflow

- Heap overflow requires modification of boundary tags
  - in-band management information
  - task is to fake these tags to trick *malloc* into overwriting addresses of attackers choice
- However, this strongly depends on the corresponding memory manager
  - They all have their implementation differences
- Often interesting information is stored on the heap
  - C++ vtables, function pointers
  - Often easier to overwrite these objects

Heap Overflow - unlink/free



- unlink:
  - `*(0x42424242+12) = 0x43434343`
- Not so easy anymore
  - `(P->fd->bk != P || P->bk->fd != P) != False`

How2Heap

- Easiest way to learn is to try yourself and look at examples.
- https://github.com/shellphish/how2heap
  - A repository for learning various heap exploitation techniques.

Heap Details

- There is much more about heaps we did not cover in depth:
  - Arenas and Binning
    - Different heaps for threads and different bins for different chunk sizes
  - Chunk coalescing
    - How free chunks are merged
- These details depended on the underlying implementation

Dangling Pointers

- A pointer that references data that is freed and which could be re-used by the program
- No guarantees can be made on the data anymore after it’s freed

Heap Spraying

- Requirement:
  - we need control over memory allocations
  - must create many objects containing shellcode
- Solution: embedded scripts
  - today, many applications allow execution of user-provided scripts in the context of the application/document to enrich usability
  - JavaScript (browsers, pdf readers)
  - ActionScript (flash applications)
- Before exploiting a memory corruption bug, allocate many objects (e.g., strings) filled with shellcode
  - It'S actually not a vulnerability, we just *use* the Heap

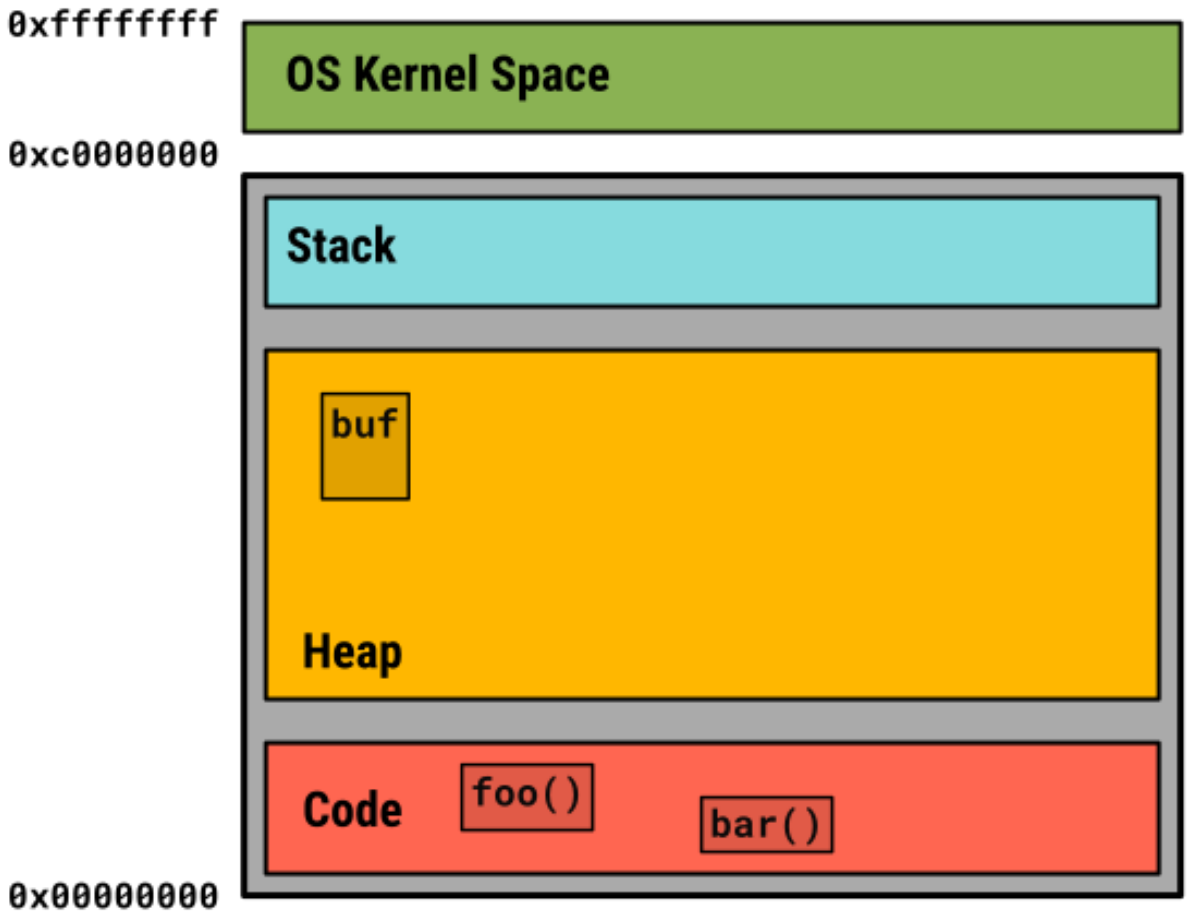
Heap Spraying - 32 vs 64 bit

- 32 bit systems have a maximum address space of 4GB
  - pretty easy to fill up
- 64 bit systems have and address space of 2^{64}
  - 18446744073709551616 bytes
  - over 18 exabytes
  - no chance to spray the full heap, but you could still do targeted spraying (e.g. if you are able to modify a heap pointer slightly)

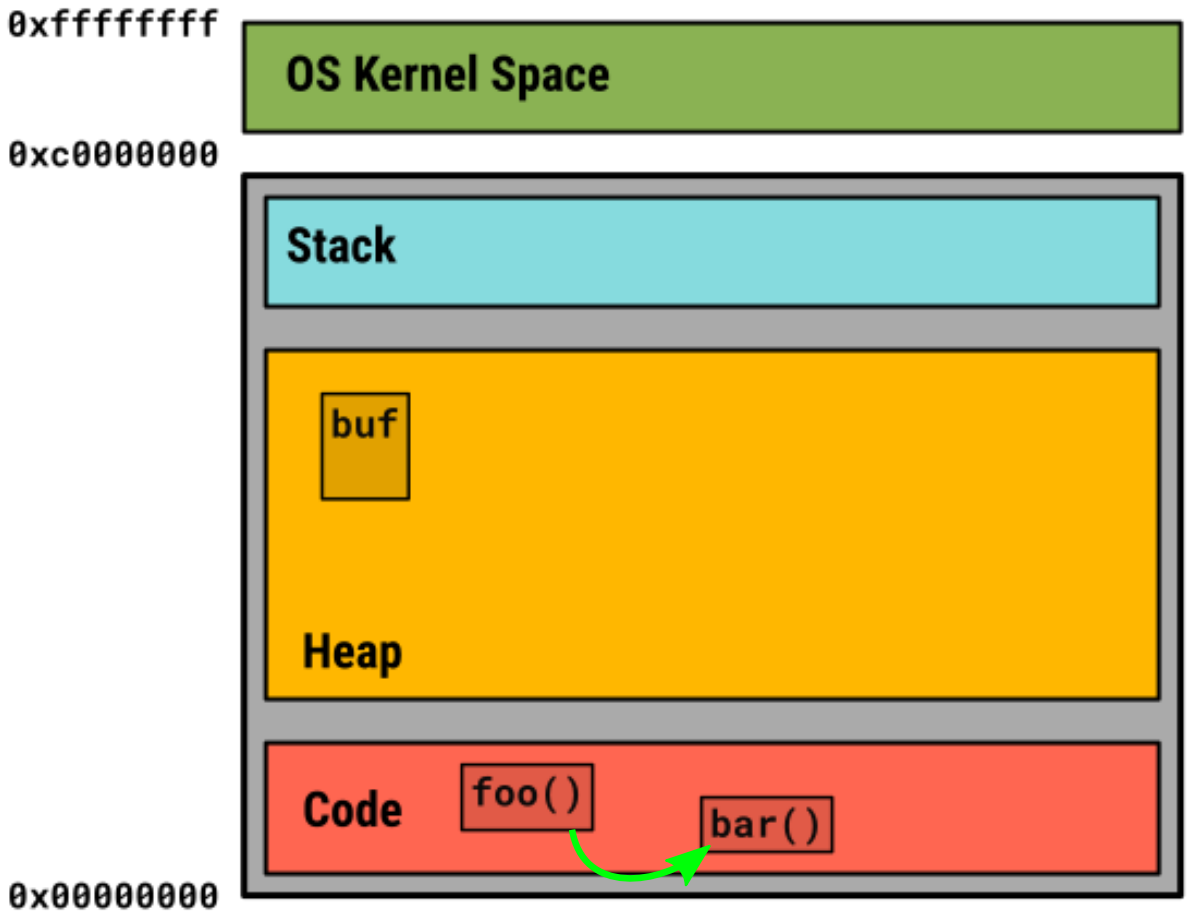
Payloads for Heap Spraying

- Previously it had been shellcodes (since heap was executable)
- Nowadays mostly fake objects or ROP chains

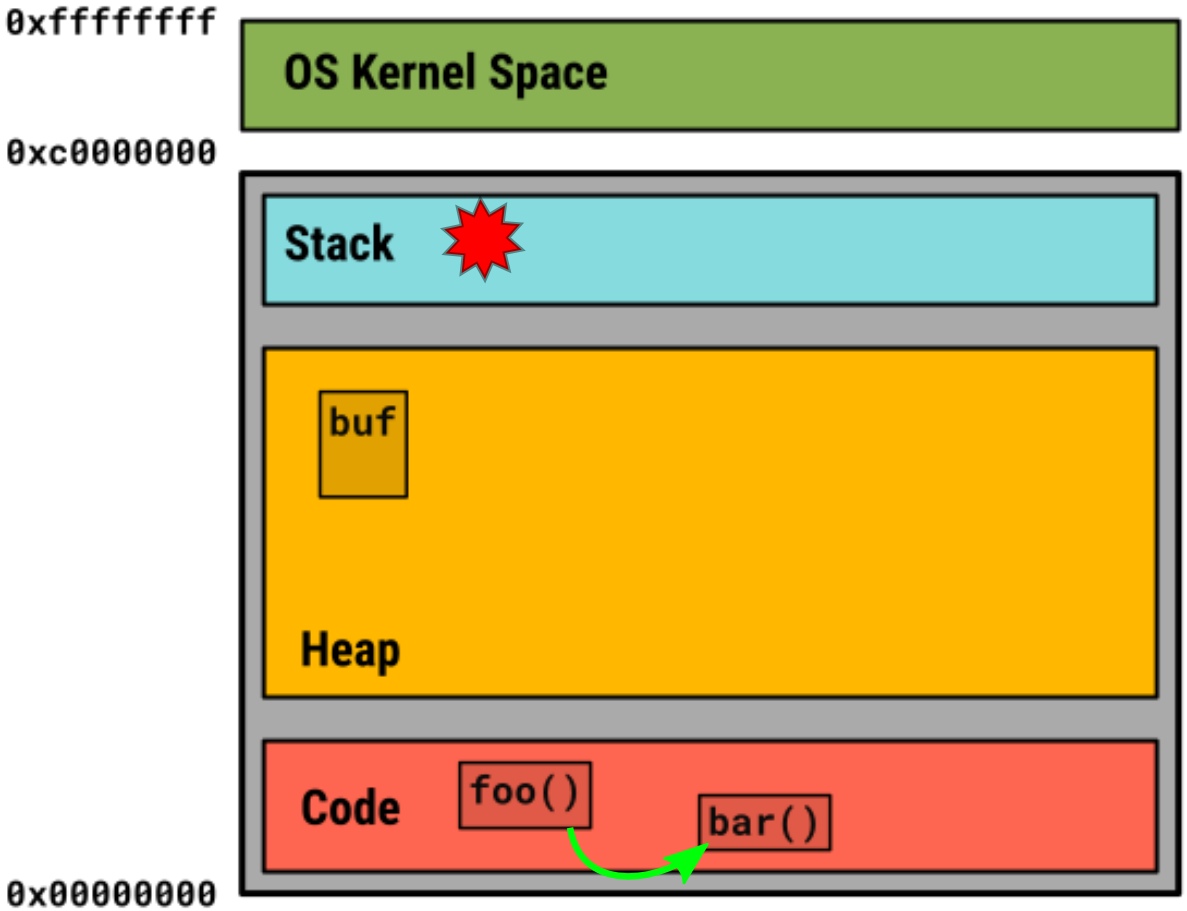
Heap Spraying



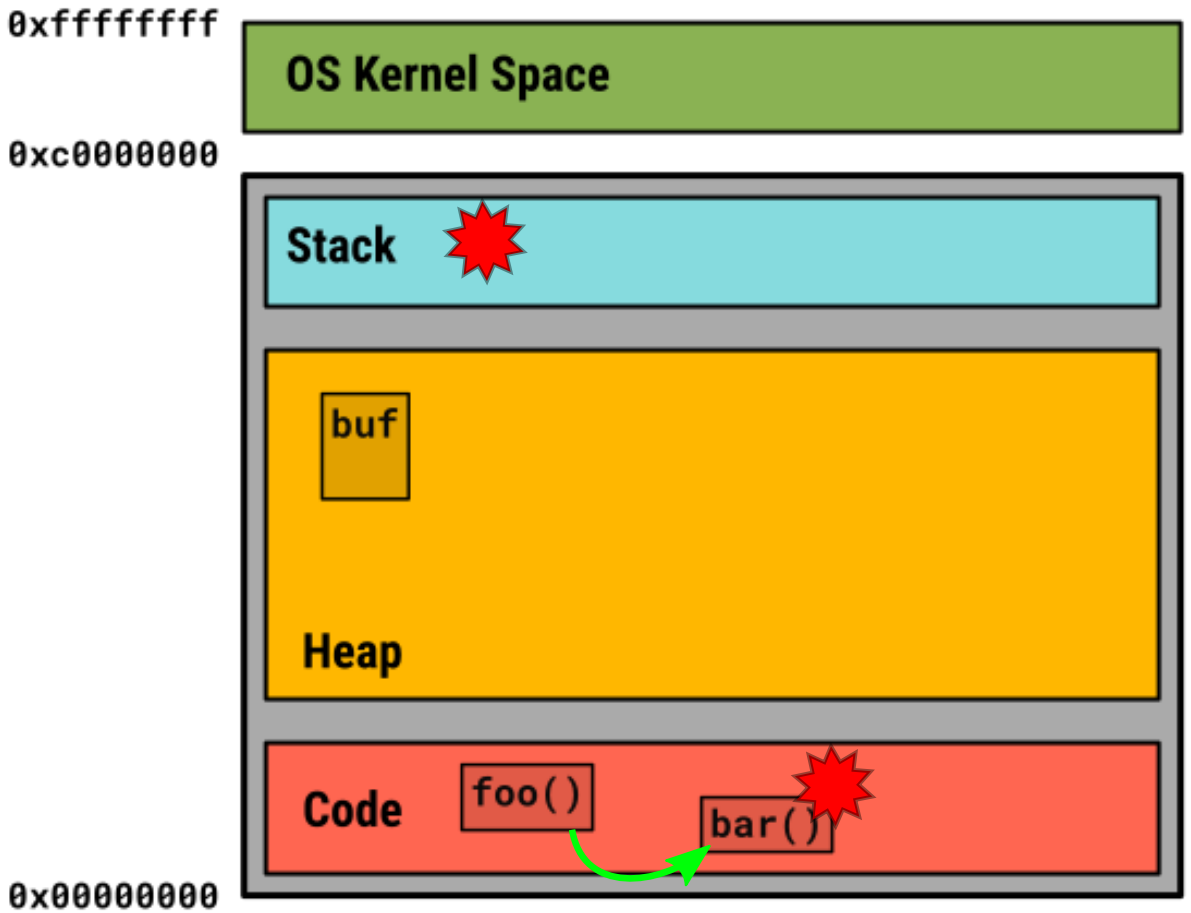
Heap Spraying



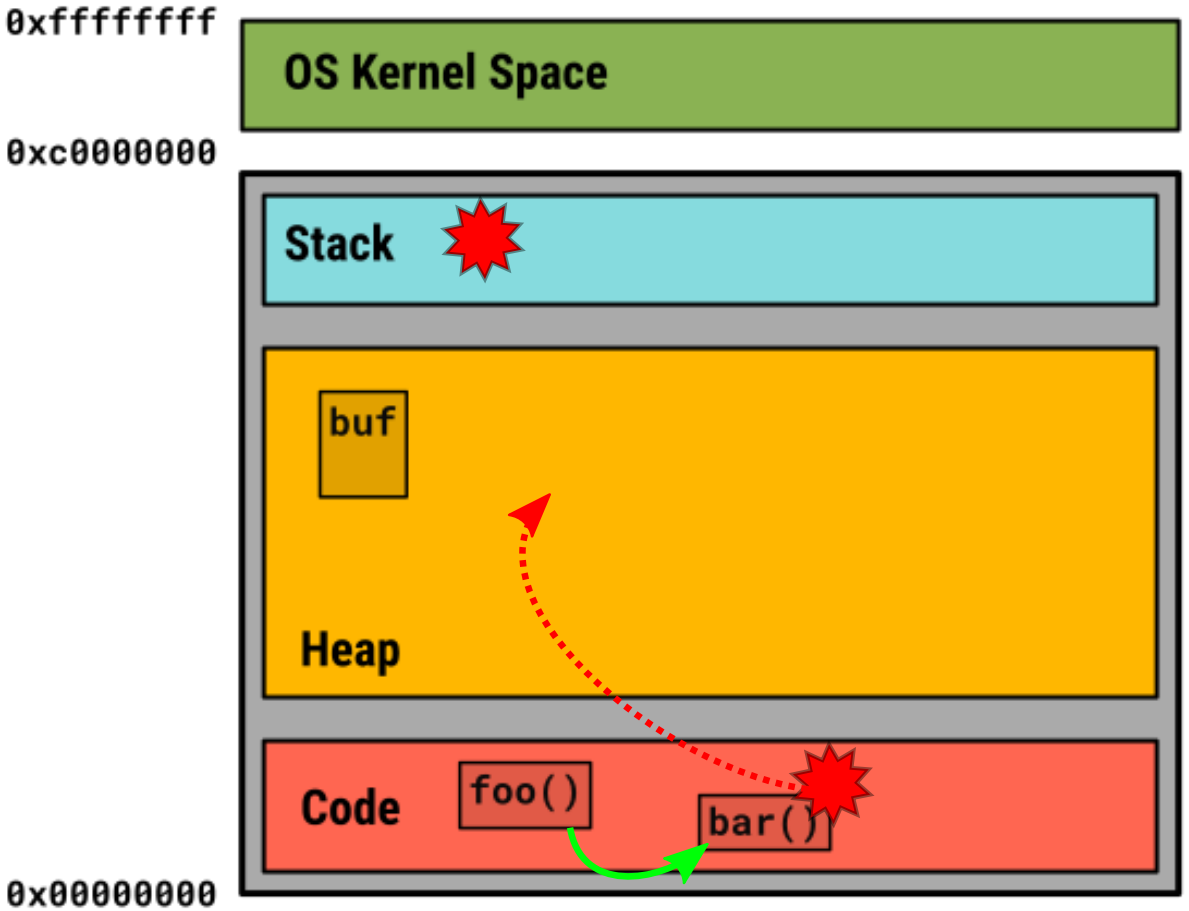
Heap Spraying



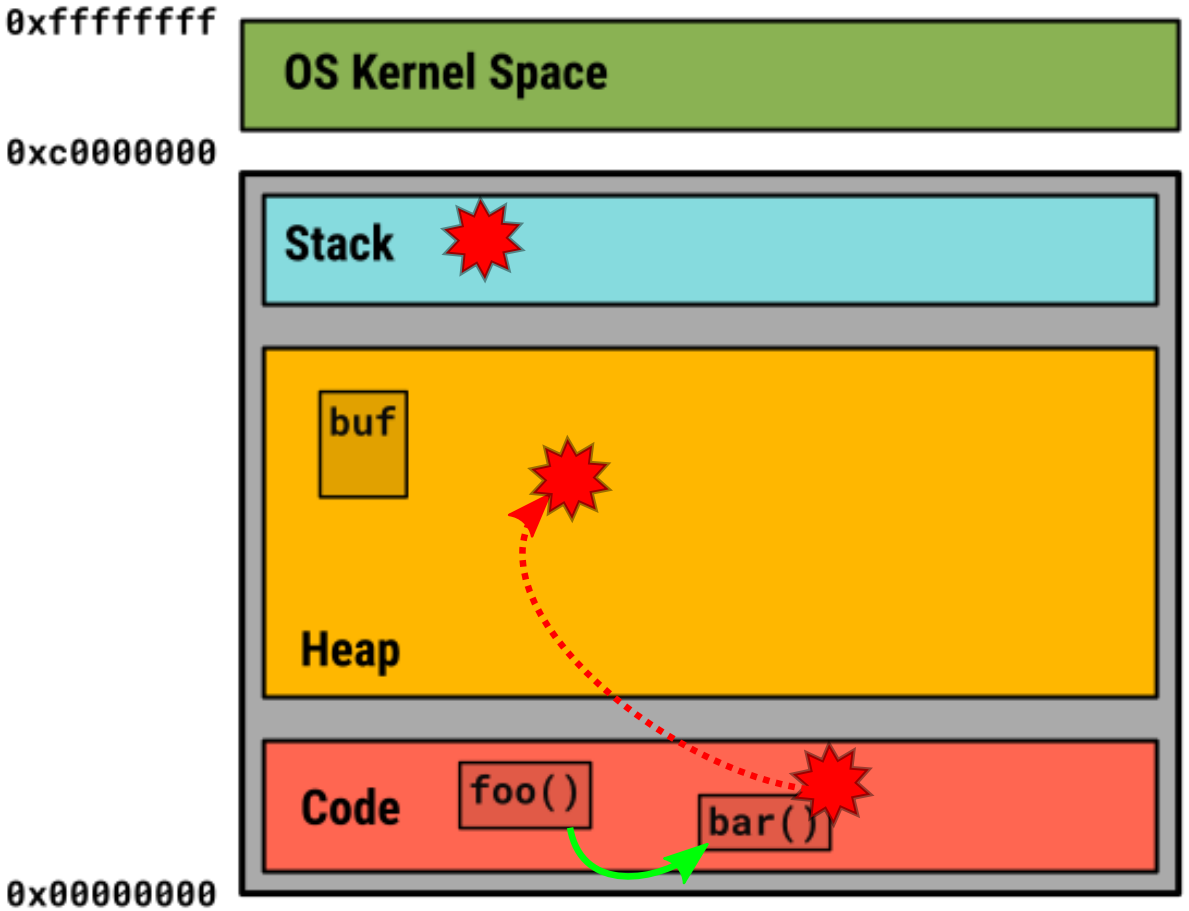
Heap Spraying



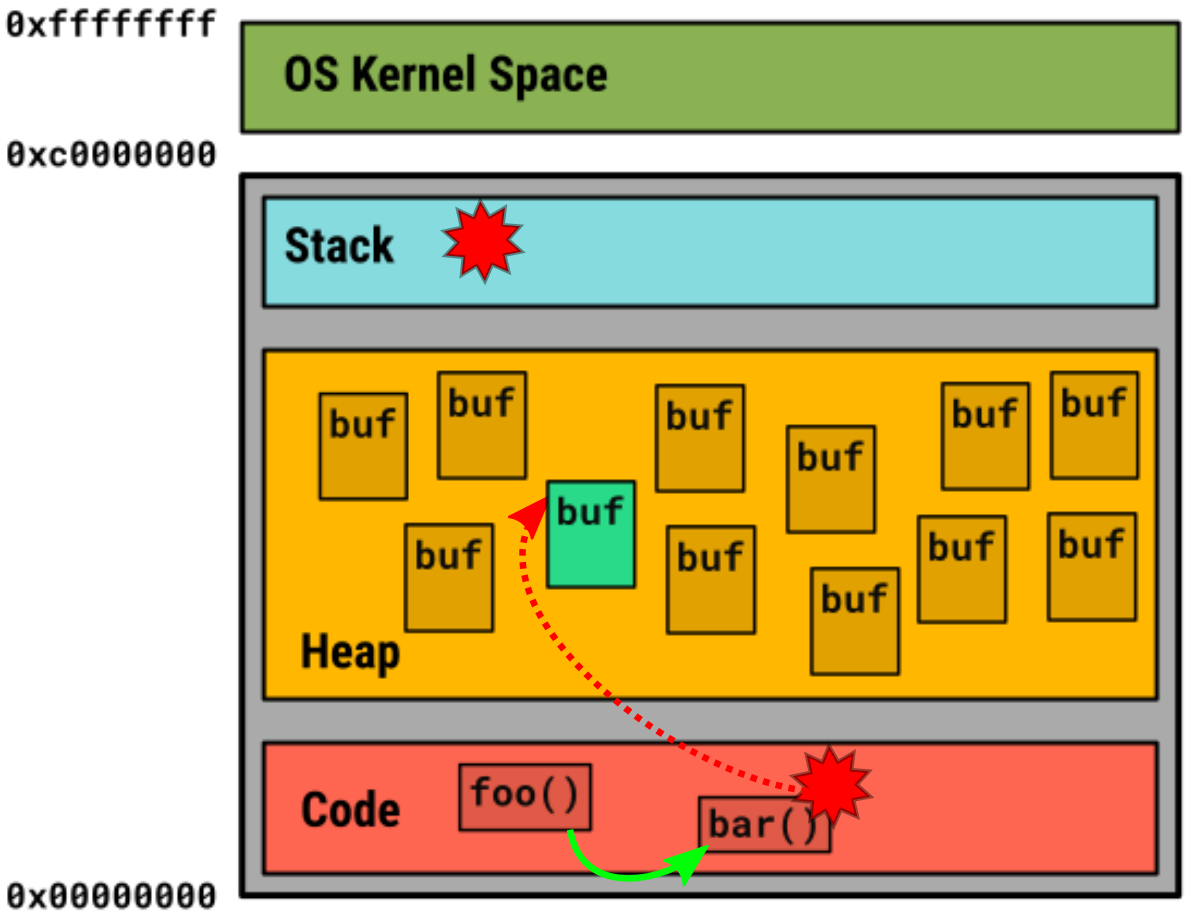
Heap Spraying



Heap Spraying



Heap Spraying



JIT Spraying

- Heap not executable
  - Cannnot just spray shellcode anymore
- JIT compilers need to create executable code on the fly
  - Spray the Heap with JIT code
    - JavaScript (Browser + PDF)
    - BPF (Kernel)
    - ...
- Code can include constants $\rightarrow$ Which could also be interpreted as code

Use after free (UAF)

- Happens when an object is free'd and then used again (dangling pointer)

```
delete X;
...
X->func();
```

- Objects are located in memory
  - free/delete release the memory to be reused
  - If we can change the content of memory between the free and the use $\rightarrow$ Win
  - Especially interesting for function pointers (e.g C++ vtables)

How to exploit UAF

- Free Object on Heap
- Create one or more smaller objects that fit into the free slot
  - Larger objects will not fit into the space
- If you can overwrite some function pointer that is reused $\rightarrow$
- Execute this function
- WIN

Other Attacks

Integer Overflows

- Simple unsigned 8 bit integer incremented

```
0x00
0x01
...
0xfe
0xff
???
```

- What happens here

Integer Overflows

- Simple unsigned 8 bit integer incremented

```
0x00
0x01
...
0xfe
0xff
0x00
0x01
...
```

Kernel Exploitation

- Kernel?
- Usually the kernel was exploited by creating the shellcode in userspace and then jumping back from kernel space to execute
- Protection techniques against this
  - SMEP and SMAP

SMEP & SMAP

- SMEP (Supervisor Mode Execution Protection)
  - allows pages to be protected from supervisor-mode instruction fetches
  - Disable execution of userland pages
- SMAP (Supervisor Mode Access Protection)
  - allows pages to be protected from supervisor-mode data accesses
  - Disable access to userland pages
    - Protect against ROP/Stack Pivoting
- Both do not prevent exploitation, they just make it harder by removing possibility to access user space data from kernel space

Kernel ASLR Bypass

- Kernel got ASLR
- To write ROP chains we need to break this
  - Can be the same as for other programs, e.g. Memory leaks
- But do we need a kernel vulnerability to leak information?

Side channels to the rescue

- Two recent papers/techniques (presented at CCS2016)
- Get the kernel's code layout by leveraging processor/hardware **features**

Two Papers at CCS

- 2 Papers at CCS
- Both leveraged hardware features
- Different Approaches Same Goal
  - Break kernel space ASLR without memory leaks

The Future?

- Darpa Cyber Grand challenge
  - https://www.youtube.com/watch?v=OVV_k73z3E0
- A challenge organised by Darpa
  - Develop autonomous systems tha
  - patches itself
  - and exploits others

Resources and Further Reading

- `https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/`
  - Understanding glibc malloc, *sploitfun*
- We touched a lot, but not everything!

Advanced Analysis Techniques

Instrumentation

- is the capability to monitor and modify program behaviour during execution
  - Source code or binary
  - Static vs dynamic
- Source/Compile-time Instrumentation
  - instrument the source code of programs
- Binary Instrumentation
  - instrument executables directly

Instrumentation Example

- Assume the following code
  - Obvious problem if len $>$ 10

```
char num[10];
for(x=0;x<len;x++) {

    num[x] = x * 2;
}
```

Instrumentation Example

- Assume the following code
  - We could add debug output

```
char num[10];
for(x=0;x<len;x++) {
    printf("%d", x);
    num[x] = x * 2;
}
```

Instrumentation Example

- Assume the following code
  - Or checks

```
char num[10];
for(x=0;x<len;x++) {
    assert(x < 10);
    num[x] = x * 2;
}
```

Source based instrumentation

- Basically every compiler can do this/does this
  - GCC
  - LLVM + Clang
  - …
- E.g adding security checks to software during compilation
- Can also be used for deeper analyses or Bug Hunting

Dynamic Binary Instrumentation

- DBI allows you to insert additional code into a binary
  - "Hooking" instructions, functions,…
  - Execute your own code before and/or after instructions
- Advantages:
  - No need to recompile or relink
  - Discover Code at runtime
  - Handle dynamically-generated code
    - You can't do this on the source
  - Attach to/instrument already running processes

DBI Use-Cases

- Instrument Binary to extract further information during runtime
  - Inject code into the binary
    - Monitor/Modify State during execution
  - Usually used for:
    - Simulation, Performance, Call graphs,…
  - We use it for:
    - Covert Debugging, Automated DeObfuscation, Taint Analysis, …

DBI Engines

- Several Engines out there
  - Pin
  - DynamoRIO
  - Valgrind
  - Frida
  - … (Many More)

Dynamic Taint Analysis

- Tracking interesting stuff (taint tracking, tainting)
- Simple idea:
  - Label information with tags
    - e.g. trusted/untrusted, interesting/boring, public/secret
  - Control how the data and labels propagate
    - When copying the data also copy the tag
- Either on binary or source level
- Can be used to check where information is propagated/used in a program

SAT/SMT Solvers

- Complex code / obfuscations
  - e.g. Obfuscated Checks of input
  - You want to know which input solves the Problem
  - Sometimes it is hard to understand/reverse the effects of the code
- You can use SAT/SMT Solvers to solve the problem for you
  - e.g. Z3 from Microsoft (there are also python bindings)
- But there is even more…

Symbolic Execution

- Based on Abstract interpretation
  - Usually binary is lifted to some intermediate representation (IR)
  - Reasoning on this abstract interpretation
  - Explore paths in parallel (beware: possible state explosion)
- We can use it to
  - identify inputs to trigger possible paths through the binary
  - or bug hunting
  - and more

Concolic Execution

- A combination between “**Conc**rete” and “Symb**olic**” Execution
  - Or dynamic symbolic execution
- Uses both techniques to solve a constraint path
  - Symbolic execution creates new concrete inputs to maximize code coverage
- Instrument Program to do symbolic execution as program runs
  - Shadow concrete program state with symbolic variables
- Explore one path at a time, start to finish
  - Can always rely on a concrete underlying value
- Can be used for Bug hunting, automatic exploit generation,…

Summary

- Format String Vulnerability
- The Heap
- Other Problems
- Advanced Analysis

Conclusion

- This concludes our lectures on Reverse Engineering and Memory Corruption
- If you liked these topics and want to do a practica/thesis, just write me a mail
- You can also take a look at our homepage for topics