Heap Models For Exploit Systems

IEEE Security and Privacy LangSec Workshop 2015

Julien Vanegue

Bloomberg L.P.
New York, USA.

May 24, 2015

Big picture : The Automated Exploitation Grand Challenge

- A **Security Exploit** is a program taking advantage of another program's vulnerability to allow untrusted code execution or obtention of secret information.
- **Automated Exploitation** is the ability for a computer to generate an exploit without human interaction.
- **The Automated Exploitation Grand Challenge** is a list of core problems in Automated Exploitation. Most (all?) problems are unsolved today for real-world cases.
- Problems relate to: Exploit Specification, Input Generation, State Space Representation, Concurrency Exploration, Privilege Inference, etc.
- The complete challenge is described at: `http://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf`

Today's topic: Heap layout prediction - AEGC Problem I

Disclaimer: this is work in progress research.

Tooling is still in development (no evaluation provided).

Presentation acts on a simplified heap.

Heap can be **non-deterministic**, we focus here on the **deterministic** heap behavior only.

Why is this an important problem?

- Nowadays, heap-based security exploits are common intrusion software.
- Exploit mitigations have made writing these exploits an expert's job.
- Heap allocator implementations are vastly different across Operating Systems.
- **There is close to no formal research on the topic**.
- Agenda: Craft and formalize a generic heap exploit technique.

Reminder: Heap vulnerability classes

- **Heap-based buffer overflow** - Overwrite adjacent memory chunk.
- **Double free / Invalid free** - Free data that is not a valid allocated chunk.
- **Use-after-free** - A pointer that was freed is cached and incorrectly used.
- **Information disclosures** - An attacker can read the content of memory.

Reminder: Heap-based buffer overflow

```
char* do_strdup(char *input, unsigned short len) {
unsigned short size = len + 1; // May overflow short capacity
char *ptr = malloc(size); // allocate small amount of memory
if (ptr == NULL)
return (NULL);
memcpy(ptr, input, len); // Buffer overflow may happen
return ptr;
}
```

Reminder: Invalid free

```
int posnum2str(int x) {
char *result;
if (x ≤ 0) goto end; // Early exit
result = calloc(20, 1);
if (result == NULL)
return (NULL);
if (num2str(result, x) == 0)
return (result);
end: free(result); // May free uninitialized pointer
return (NULL);
}
```

Reminder: Use-after-free

```
char *compute(int sz) {
char *ptr = malloc(sz);
if (ptr == NULL) return (NULL);
int len = f(ptr); // Assume f will free ptr under some conditions
ptr[len] = 0x00; // ptr was already freed!
return (ptr);
}
```

Reminder: Information disclosure

```
Require: sock : Valid network socket
Ensure: True on success, False on failure
char buff[MAX_SIZE]
int readlen = recv(sock, buff, MAX_SIZE);
if (readlen ≤ 0) return False;
rec_t *hdr = (rec_t *) buff;
char *out = malloc(sizeof(rec_t) + hdr->len);
if (NULL == out) return (false);
memcpy(out, buff + sizeof(rec_t), hdr->len); // Read out of bound
out->len = hdr->len;
send(sock, out, hdr->len + sizeof(rec_t)); // Send memory to attacker
free(out);
return True
```

Original AEGC problem I harness test

```
struct s1 { int *ptr; } *p1a = NULL, *p1b = NULL, *p1c = NULL;
struct s2 { int authenticated; } *p2 = NULL;
F() {
p1a = (struct s1*) calloc(sizeof(struct s1), 1);
p1b = (struct s1*) calloc(sizeof(struct s1), 1);
p1c = (struct s1*) calloc(sizeof(struct s1), 1);
}
G() { p2 = (struct s2*) calloc(sizeof(struct s2), 1); }
H() { free(p1b); }
I() { memset(p1a, 0x01, 32); } // Buffer overflow
J() { if (p2 && p2->authenticated) puts(you win); } // Go here
K() { if (p1a && p1a->ptr) *(p1a->ptr) = 0x42; } // Avoid crash
```

Goal: Automate heap walk = { F(); H(); G(); I(); J(); }

What do these vulnerabilities have in common?

- In heap overflow case, attacker expects to place an interesting chunk after the overflowed chunk.
- In use-after-free case, attacker expects to place controlled chunk in freed memory before it is used incorrectly.
- In invalid free case, attacker expects to place controlled heap memory at location of invalid free.
- In information disclosure, attacker expects to place secret in heap just after chunk allowing disclosure.
- In harness test of Problem I (previous slide), we expect chunk p2 to be reusing p1b's memory after it was freed.
- Summing up: Exploitation depends on location of chunks relative to each others.
- What is a good **layout abstraction** for the heap?

Studied allocators

- Doug Lea's malloc (DLMalloc) - Linux.
- PTMalloc (DLMalloc + thread support) - Linux.
- Windows heap (including Low Fragmentation Heap).
- NOT studied: JEmalloc (FreeBSD / NetBSD / Firefox).
- NOT studied: Garbage Collection (Sweep and Mark algorithm etc).

Typical (simplified) heap allocation algorithm

1. Try to use one of the cached (last freed) chunks.
2. Try to find a fitting chunk in the free chunks list.
3. Try to coallesce two free chunks from free list.
4. If still fails, try (2,3) with each free list in increasing order.
5. If everything fails, try to extend the heap.
6. Otherwise, return an error (NULL)

Formal heap definition

$\mathcal{H} = (\mathcal{L}, \Gamma_a, \Gamma_f, ADJ, Top)$ where:

- $\mathcal{L} = (l_1, l_2, ..., l_n)$ is a set of lists of available memory chunks. Each list holds free chunks for a given size range.
- $l = (c_1, c_2, ..., c_n)$ are individual memory chunks in list l.
- $\Gamma_a : l \rightarrow int$ is a **counter map of allocated chunks** for a given size range.
- $\Gamma_f : l \rightarrow int$ is a **counter map of free chunks** for a given size range.
- $ADJ : c \times c \rightarrow \mathcal{B}$ is the **adjacency** predicate (true if chunks are immediately adjacent).
- Top is the current chunk in $\mathcal{H}$ with the highest address.

Heap semantics

Heap primitives:

(F)ree : A memory chunk is freed.
(R)ealloc : A memory chunk is extended.
(A)lloc : A memory chunk is allocated.
(C)oallesce : Two memory chunks are merged.
(S)plit : A big memory chunk is split into two smaller ones.
(E)xtend : The heap is extended by a desired size

Heap transition system:

$$\mathcal{H}' \longleftarrow \text{F p } \mathcal{H}$$
$$(\mathcal{H}', p2) \longleftarrow \text{R p1 sz } \mathcal{H}$$
$$(\mathcal{H}', p) \longleftarrow \text{A sz } \mathcal{H}$$
$$(\mathcal{H}', p3) \longleftarrow \text{C p1 p2 } \mathcal{H}$$
$$(\mathcal{H}', p2, p3) \longleftarrow \text{S p1 off } \mathcal{H}$$
$$(\mathcal{H}', p) \longleftarrow \text{E sz } \mathcal{H}$$

Key ideas

1. There are two levels of semantics: physical and logical:
   - The **physical semantic** is concerned with the adjacency of chunks in memory.
   - The **logical semantic** is concerned with the population of chunk lists.
   - **Our goal is to reconcile physical and logical heap semantics**.
2. Heap primitives must include user interactions (F, R, A).
3. Core internal heap mechanisms are defined as first class primitives (C, S, E).
4. An Adjacency predicate **ADJ** (used in S and E only) defines the physical semantic. Everything else is house cleaning and defines the logical semantic using two counters per list.
5. Defining the heap transition system allows us to reduce the problem to a reachability algorithm.

Prerequisite: Heap **List Fitness** algorithm (here *best fit* in ML-style syntax)

```
let best (cur:Chunk)(sz:int)(cand:Chunk) =
if (cur.size ≤ sz and cur.sz - sz ≤ cand.sz - sz)
then cur else cand;;

let rec findfit (choice: a → b → c → d)(l:list)(sz:int)(cand:Chunk) in
match l with
| [] → cand
| [cur::tail] → (findfit tail sz (choice cur size cand));;

let rec FIT Lists sz = match Lists with
| [] → ⊥
| [cur::tail] → let res = (findfit best cur sz ⊥) in
                match res with
                |⊥ → (fit tail sz)
                | cur;;
```

The FRACSE calculus (part 1)

$$\dfrac{\dfrac{size(p) = x \qquad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x) = l_1}{\mathbf{FREE}(p)}}{\Gamma'_a[l_1] \leftarrow \Gamma_a[l_1] - 1 \qquad \Gamma'_f[l_1] \leftarrow \Gamma_f[l_1] + 1}$$

$$\dfrac{\dfrac{\mathbf{FIT}(\mathcal{H}.\mathcal{L}, x) = l_1}{p = \mathbf{ALLOC}(x)}}{\Gamma'_a[l_1] \leftarrow \Gamma_a[l_1] + 1 \qquad \Gamma'_f[l_1] \leftarrow \Gamma_f[l_1] - 1}$$

$$\dfrac{\dfrac{size(p) = x \qquad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x) = l_1 \qquad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x + e) = l_2}{p_2 = \mathbf{REALLOC}(p_1, x + e)}}{\Gamma'_a[l_1] \leftarrow \Gamma_a[l_1] - 1 \qquad \Gamma'_f[l_1] \leftarrow \Gamma_f[l_1] + 1 \qquad \Gamma'_a[l_2] \leftarrow \Gamma_a[l_2] + 1 \qquad \Gamma'_f[l_2] \leftarrow \Gamma_f[l_2] - 1}$$

The FRACSE calculus (part 2)

$$\frac{\dfrac{size(p_1) = x_1 \quad size(p_2) = x_2 \quad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x_1) = l_1 \quad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x_2) = l_2 \quad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x_3) = l_3}{p_3 = \mathbf{COALLESCE}(p_1, p_2)}}{\Gamma'_f[l_1] \leftarrow \Gamma_f[l_1] - 1 \quad \Gamma'_f[l_2] \leftarrow \Gamma_f[l_2] - 1 \quad \Gamma'_f[l_3] = \Gamma_f[l3] + 1}$$

$$\frac{\dfrac{size(p) = x \quad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x) = l_1 \quad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, x - o) = l_2 \quad \mathbf{FIT}(\mathcal{H}.\mathcal{L}, o) = l_3}{(p1, p2) = \mathbf{SPLIT}(p, o)}}{\mathbf{ADJ}(p_1, p_2) \quad \Gamma'_f[l_1] \leftarrow \Gamma_f[l_1] - 1 \quad \Gamma'_f[l_2] \leftarrow \Gamma_f[l_2] + 1 \quad \Gamma'_f[l_3] \leftarrow \Gamma_f[l_3] + 1}$$

$$\frac{\dfrac{\mathbf{FIT}(\mathcal{H}.\mathcal{L}, x) = l}{p = \mathbf{EXTEND}(x)}}{\mathbf{ADJ}(Top, p) \quad \Gamma'_f[l] \leftarrow \Gamma_f[l] + 1 \quad Top \leftarrow p}$$

Pitfalls

- There can be multiple heaps (ex: one per thread). Heap selection is not defined in the FRACSE semantics. As **FIT** uses a heap parameter, it can handle multiple heaps easily.
- There can be multiple allocators within a process (ex: Windows front-end / back-end) driven by an **activation heuristic** for each bucket size. Adding such activation heuristic is a reasonable extension.
- FRACSE uses *lists*, some allocators use *arrays* (ex: JEMalloc)
- Heap meta-data is **abstracted** by design. Some exploit techniques still rely on meta-data corruption. We argue that due to internal checks in allocators, heap meta-data corruption as an exploit technique is dying.
- Non-deterministic heap behavior is not covered (ex: Die Hard allocator randomization, LFH subsegment randomization, etc). We need a probabilistic semantics to define this.
- This presentation only covers user-land heap allocators, no kernel heap allocator.

Summing up

- This work may be the first attempt at defining the formal semantics of heap allocators.
- Heap allocator implementations are so different that making generic heap analysis is a challenge.
- However, we can distinguish some common functionalities (split/coallesce/extend operations, list-based abstraction, heap selection, etc).
- Focusing on targeting user data and using a heap layout abstraction seems like the only generic way of exploiting the heap.
- FRACSE implementation is still going on. Its calculus may evolve based on experiments.

Thanks for attending!

Questions?

Mail: julien.vanegue@gmail.com
Twitter: @jvanegue

(Some) Related work

1. Smashing C++ VPTRS (Eric Landuyt)
2. VuDo Malloc tricks (Michel Kaempf)
3. Once upon a free (Scut)
4. Advanced DLMalloc Exploits (JP)
5. Malloc Maleficarum (Phantasmal Phantasmagoria)
6. The use of set_head to defeat the wilderness (g463)
7. Heap Feng Shui (Alex Sotirov)
8. Understanding the Low Fragmentation Heap (Chris Valasek)
9. The House Of Lore : PTmalloc exploitation (blackngel)
10. Pseudomonarchia Jemallocum (argp and huku)