Our Blog

Our news

All you need to know

2017 (25) 2016 (22) 2015 (17) 2014 (16) 2013 (30) 2012 (27) 2011 (33)

Linux Heap Exploitation Intro Series: Used and Abused – Use After Free

Reading time ~9 min

Posted by javier on 28 July 2017

Categories: Heap, Heap linux, Pwnage friday, Expl

Page Saved!

Add Tags

Intro

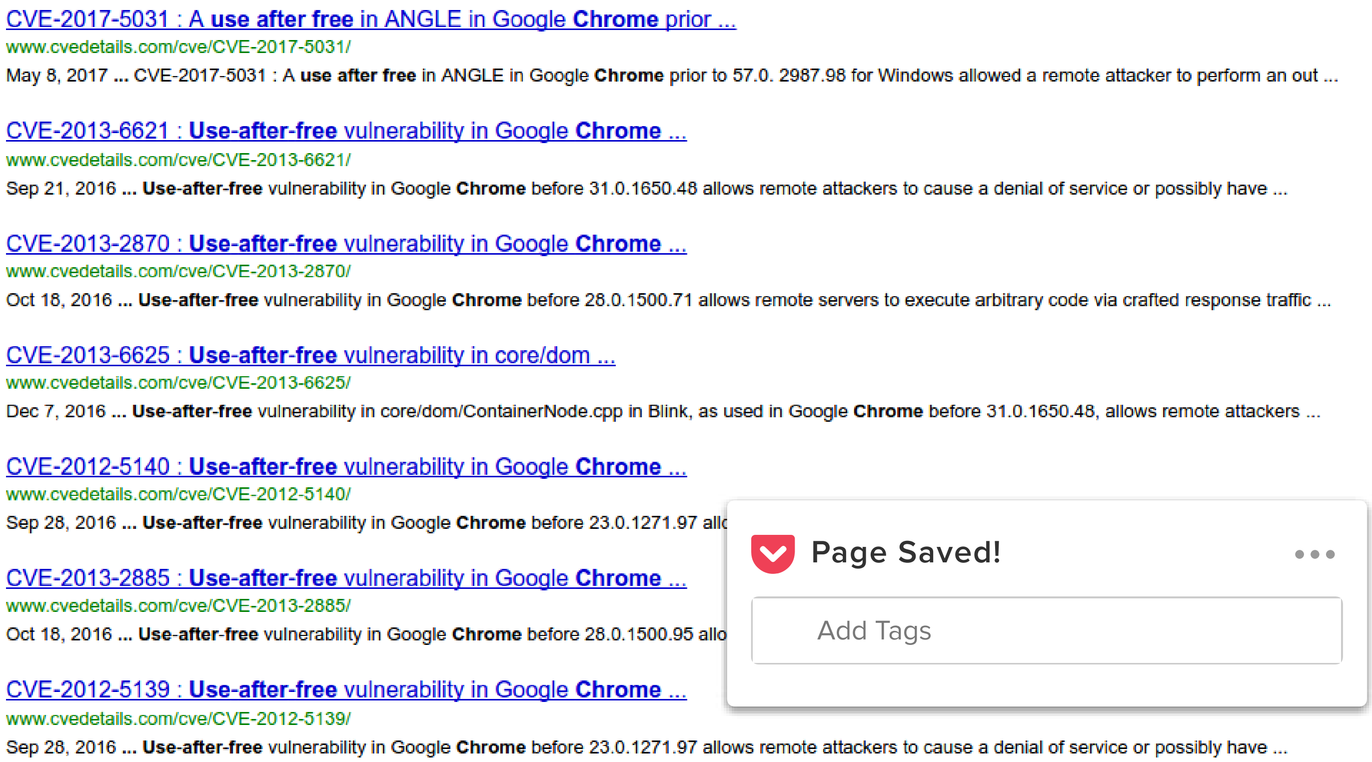
After analysing the implementation of ptmalloc2 which, **is a must read if you don't know anything about the linux userland heap**, I decided that for the second part of it, I would approach it as in a series of blog posts. Why? You might ask. Well it is easy for someone to tackle a problem in bite sized "chunks". Understanding the heaps can be difficult and each of the techniques to be described in this series takes a decent amount of time to learn, understand and practice. Also, it is easier to find 15 minutes in a day rather than a few hours in a day. Also – hack the system.

Please note that it is not needed, but having a rough knowledge of programming and how pointers work in **C** code is a great shortcut to understand the examples given in this series.

The Vulnerability

Preface

To start this series we are going to cover one specific type of **use-after-invalidation** vulnerability because of its fame and presence. Going through www.cvedetails.com and searching for two of the most used browsers (Chrome and Firefox) yields many CVE's involving this specific vulnerability: The *use-after-free*.


CVE-2017-5031 : A **use after free** in ANGLE in Google **Chrome** prior ...
www.cvedetails.com/cve/CVE-2017-5031/
May 8, 2017 ... CVE-2017-5031 : A **use after free** in ANGLE in Google **Chrome** prior to 57.0. 2987.98 for Windows allowed a remote attacker to perform an out ...

CVE-2013-6621 : **Use-after-free** vulnerability in Google **Chrome** ...
www.cvedetails.com/cve/CVE-2013-6621/
Sep 21, 2016 ... **Use-after-free** vulnerability in Google **Chrome** before 31.0.1650.48 allows remote attackers to cause a denial of service or possibly have ...

CVE-2013-2870 : **Use-after-free** vulnerability in Google **Chrome** ...
www.cvedetails.com/cve/CVE-2013-2870/
Oct 18, 2016 ... **Use-after-free** vulnerability in Google **Chrome** before 28.0.1500.71 allows remote servers to execute arbitrary code via crafted response traffic ...

CVE-2013-6625 : **Use-after-free** vulnerability in core/dom ...
www.cvedetails.com/cve/CVE-2013-6625/
Dec 7, 2016 ... **Use-after-free** vulnerability in core/dom/ContainerNode.cpp in Blink, as used in Google **Chrome** before 31.0.1650.48, allows remote attackers ...

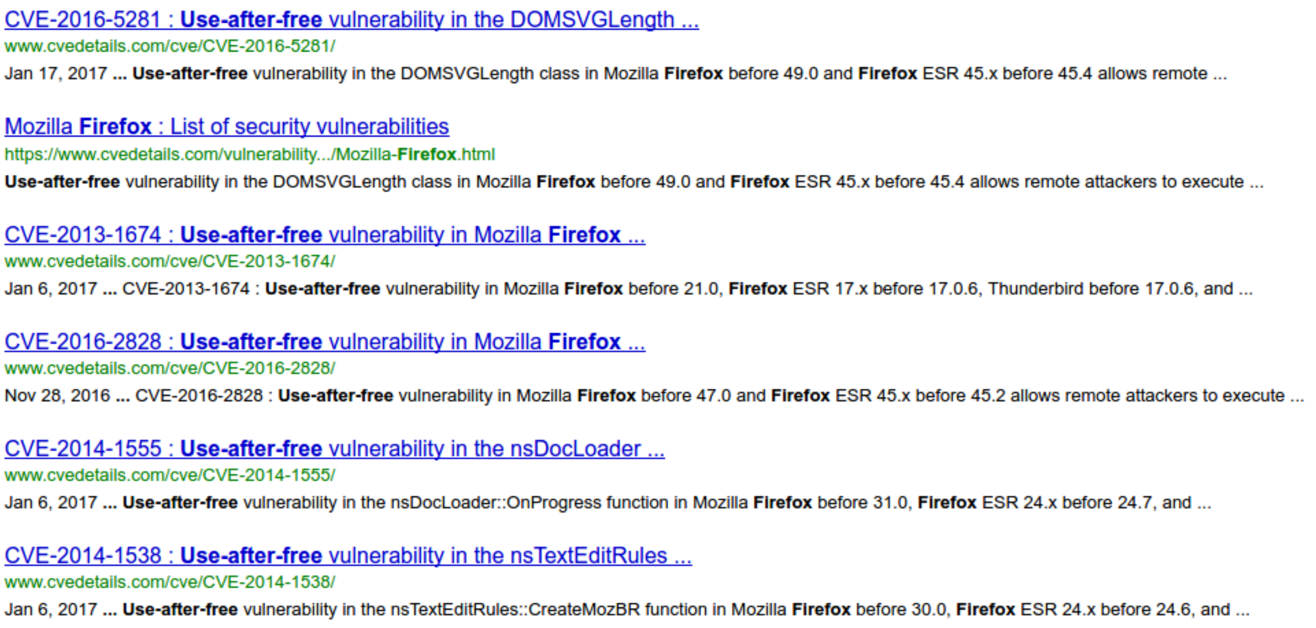
CVE-2012-5140 : **Use-after-free** vulnerability in Google **Chrome** ...
www.cvedetails.com/cve/CVE-2012-5140/
Sep 28, 2016 ... **Use-after-free** vulnerability in Google **Chrome** before 23.0.1271.97 allc

CVE-2013-2885 : **Use-after-free** vulnerability in Google **Chrome** ...
www.cvedetails.com/cve/CVE-2013-2885/
Oct 18, 2016 ... **Use-after-free** vulnerability in Google **Chrome** before 28.0.1500.95 allo

CVE-2012-5139 : **Use-after-free** vulnerability in Google **Chrome** ...
www.cvedetails.com/cve/CVE-2012-5139/
Sep 28, 2016 ... **Use-after-free** vulnerability in Google **Chrome** before 23.0.1271.97 allows remote attackers to cause a denial of service or possibly have ...

Page Saved!
Add Tags

Chrome use-after-free vulnerabilities



CVE-2016-5281 : **Use-after-free** vulnerability in the DOMSVGLength ...
www.cvedetails.com/cve/CVE-2016-5281/
Jan 17, 2017 ... **Use-after-free** vulnerability in the DOMSVGLength class in Mozilla **Firefox** before 49.0 and **Firefox** ESR 45.x before 45.4 allows remote ...

Mozilla **Firefox** : List of security vulnerabilities
https://www.cvedetails.com/vulnerability.../Mozilla-**Firefox**.html
Use-after-free vulnerability in the DOMSVGLength class in Mozilla **Firefox** before 49.0 and **Firefox** ESR 45.x before 45.4 allows remote attackers to execute ...

CVE-2013-1674 : **Use-after-free** vulnerability in Mozilla **Firefox** ...
www.cvedetails.com/cve/CVE-2013-1674/
Jan 6, 2017 ... CVE-2013-1674 : **Use-after-free** vulnerability in Mozilla **Firefox** before 21.0, **Firefox** ESR 17.x before 17.0.6, Thunderbird before 17.0.6, and ...

CVE-2016-2828 : **Use-after-free** vulnerability in Mozilla **Firefox** ...
www.cvedetails.com/cve/CVE-2016-2828/
Nov 28, 2016 ... CVE-2016-2828 : **Use-after-free** vulnerability in Mozilla **Firefox** before 47.0 and **Firefox** ESR 45.x before 45.2 allows remote attackers to execute ...

CVE-2014-1555 : **Use-after-free** vulnerability in the nsDocLoader ...
www.cvedetails.com/cve/CVE-2014-1555/
Jan 6, 2017 ... **Use-after-free** vulnerability in the nsDocLoader::OnProgress function in Mozilla **Firefox** before 31.0, **Firefox** ESR 24.x before 24.7, and ...

CVE-2014-1538 : **Use-after-free** vulnerability in the nsTextEditRules ...
www.cvedetails.com/cve/CVE-2014-1538/
Jan 6, 2017 ... **Use-after-free** vulnerability in the nsTextEditRules::CreateMozBR function in Mozilla **Firefox** before 30.0, **Firefox** ESR 24.x before 24.6, and ...

Firefox use-after-free vulnerabilities

It is one of the most common vulnerabilities, if not the most, which is involved in heap exploitation, and it is the most likely to end up in arbitrary code execution from an attacker's perspective.g

An example of the fame and wide presence of suc
2017-8540 found by Ian Beer @ Google's Project Z

Page Saved!
Add Tags

What

The *use-after-free* vulnerability is a use-after-invalidation vulnerability[1] where **free** is the invalid state of use. In human-readable language this means that at some point of the implementation there was a logic flaw that caused a *free()* on a chunk, but despite being free()'d, its memory position is still referenced, effectively making use of the free'd chunk's data after it has been set free.

The implications of this type of vulnerability, and this is a cliché phrase when talking about heap exploitation, are endless; Pointer dereference leading to arbitrary *write-what-where*, leaking memory to defeat memory address randomisation (ASLR), dangling pointers, etc. And in some critical cases, pointer dereferences to functions. Let's see some examples!

When stars align

following:

1 – Allocate **MALLOC1**: First an allocation is done in which we provide our own structure (**pointer_malloc1**) that has a function pointer allocated in the heap.

```
 +---HEAP GROWS UPWARDS
 |                  +--- pointer_malloc1->good_function()
 | +-+-+-+-+-+-+    |
 | | MALLOC 1  |<--+
 | +-----------+
 | |    TOP    |
 | |           |
 V |           |
   +-----------+
--- WILDERNESS ---
```

2 – Free **MALLOC1**: free() the memory occupied by our first **MALLOC1**.

```
 +---HEAP GROWS UPWARDS
 |                  +--- pointer_malloc
 | +-+-+-+-+-+-+    |
 | |  FREE 1   |<--+
 | +-----------+
 | |    TOP    |
 | |           |
 V |           |
   +-----------+
--- WILDERNESS ---
```

Page Saved!
Add Tags

3 – Allocate a malicious chunk **MALLOC2** of the same size as **MALLOC1**: As we know from the previous post, chunks will be allocated by size and so, for our malicious chunk to take the place of old **MALLOC1** it has to be of the same size.

```
 +---HEAP GROWS UPWARDS
 |                  +--- pointer_malloc1->good_function()
 | +-+-+-+-+-+-+    |
 | | MALLOC 2  |<--+--- pointer_malloc2->evil_function()
 | +-----------+
 | |    TOP    |
 | |           |
 V |           |
```

4 – Reference old **MALLOC1** (by mistake): Now that we allocated our own payload inside the **MALLOC2** we expect the program's implementation to call **pointer_malloc1->good_function()** which is actually pointing to our malicious **MALLOC2** and so, it will actually call our **pointer_malloc2->evil_function()** because of the "short circuit" happening at (1).

```
 +---HEAP GROWS UPWARDS
 |                   +--- pointer_malloc1->good_function()
 | +-+-+-+-+-+-+    |
 | | MALLOC 2 |<-(1)-- pointer_malloc2->evil_function()
 | +-----------+
 | |    TOP    |
 | |           |
 V |           |
   +-----------+
--- WILDERNESS ---
```

The avid and experienced reader could point out something that my colleague Seif pointed out: This is a pointer dereference called d pointer is then used it becomes a use-after-free.

Page Saved!
Add Tags

The playground

Now on to the fun! Three snippets of code are provided here: Two of them exploiting the lack of security checks on ptmalloc2's implementation (those security checks are left for the programmer to implement in order to keep ptmalloc2 performance at an acceptable level) and a third one left as a simple exercise for the reader to exploit a use-after-free vulnerable application.

Download Playground Code

Proof of Concepts

Basic UAF 1

translating the following code to our target scripting environment (For example, JavaScript). But let's not dwell into complex stuff yet as simplicity is key to understand the following.

In the beginning of the code I have defined a **struct** which is a composite data type. This means it can contain different types of data within itself, even other structs. In our case we have a simple struct containing a pointer to a function that returns nothing (i.e. returns **void**). This struct is the one that we will be (ab)using freely, specifically the "**vulnfunc**" pointer.

```
typedef struct UAFME {
 void (*vulnfunc)();
} UAFME;
```

Afterwards, we have global definitions for two functions that simply print two strings to the terminal. Our goal is to call **bad()** without explicitly initialising any pointers to it.

```
void good(){
 printf("I AM GOOD :)\n");
}

void bad(){
 printf("I AM BAD >:|\n");
}
```

Page Saved!
Add Tags

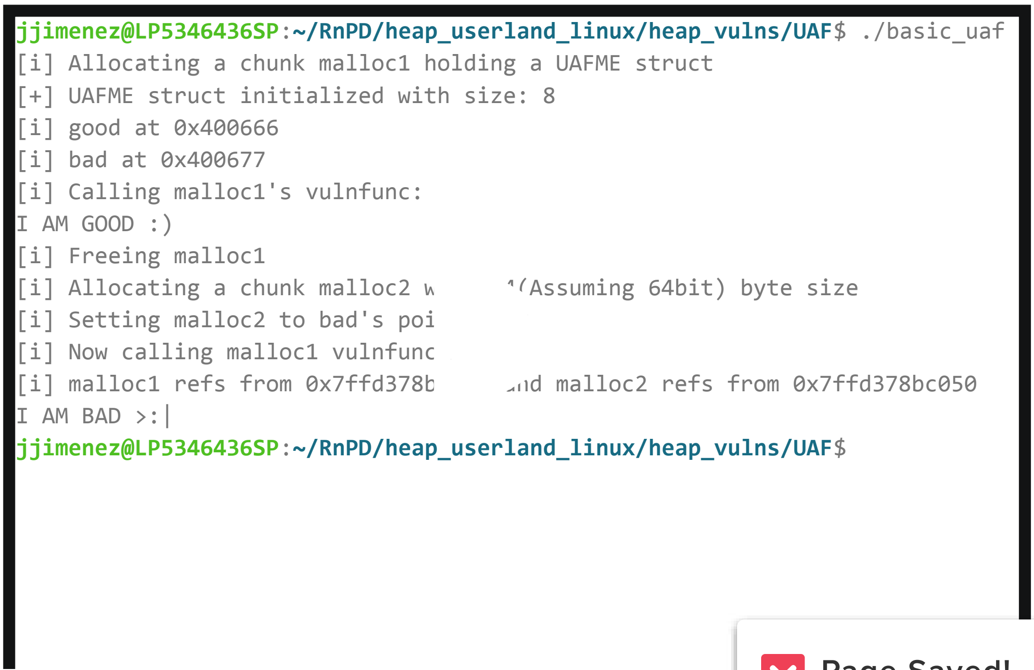
The next is an excerpt of **main** and it involves the allocation in memory of a chunk containing a pointer (1) to a struct of the aforementioned type: **UAFME**. The **vulnfunc** of said UAFME struct is set to the **good** function (2) so that when we call **malloc1->vulnfunc()**, the good function will be called (3).

```
UAFME *malloc1 = malloc(sizeof(UAFME)); // (1)
malloc1->vulnfunc = good; // (2)
malloc1->vulnfunc(); // (3)
```

Finally, we arrive to the exploitation part. Due to ptmalloc2 not checking the validity of the memory state of each variable, we can call **malloc1->vulnfunc()** again **(1)** even if its pointer has been free()'d **(2)** and so, it will actually use our malicious allocation **(3)** that we specifically craft by setting its contents to the pointer of **bad()** function **(4)**.

```
*malloc2 = (long)bad; // (4)
malloc1->vulnfunc(); // BOOM! (1)
```

Let's see it in action!



00:00

Page Saved!
Add Tags

As you can see, it does print "**I AM BAD >:|**" at the end of the execution! This happens because we are using **malloc1**'s pointer to call **vulnfunc** which actually is the pointer to the contents of **malloc2**. Eeeeviiiil.

Basic UAF 2

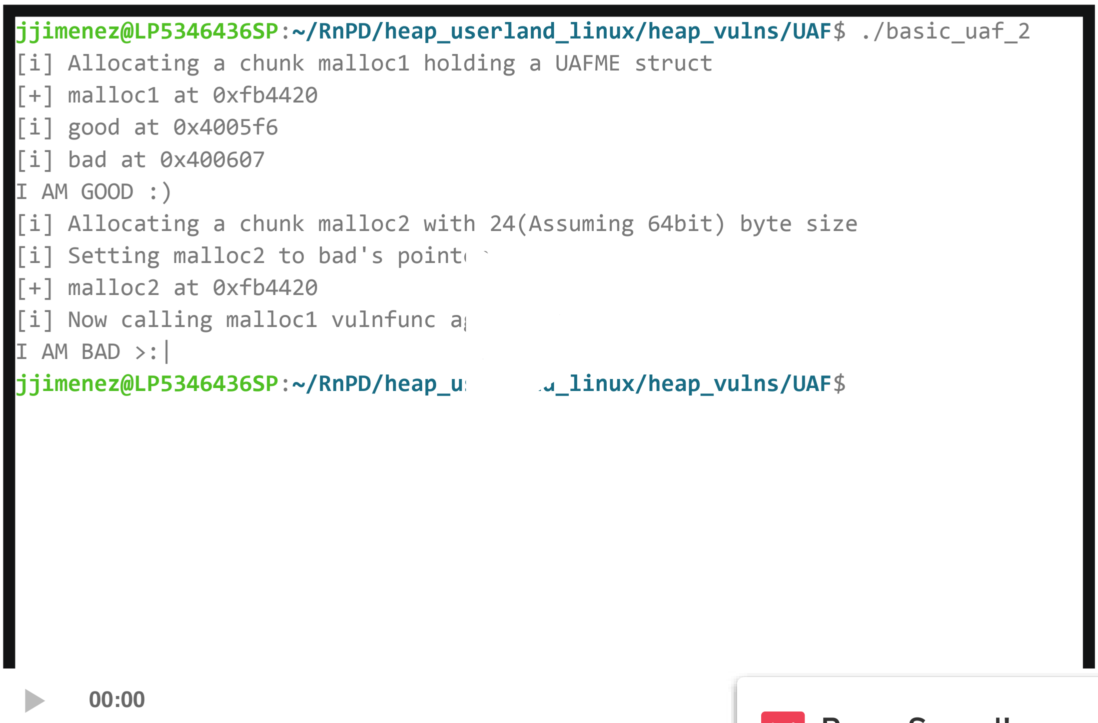
Now on a little bit more of a realistic scenario but keeping the same implementation, we are introducing a helper allocator function. On some software, programmers implement their own "safe" wrappers on top of **malloc** to prevent some vulnerabilities such as buffer overflows and unsafe **free()**s.

In our proof of concept, our developer tried to be tidy but, after a tiring day of work and a few hundred lines of code he forgot about something:

```
void helper_call_goodfunc(UAFME *uafme){
 UAFME *private_uafme = uafme;
 private_uafme->vulnfunc = good;
```

```
}
```

Can you spot it? Spoiler: The solution is in the comments of the file *basic_uaf_2.c*.



Page Saved!
Add Tags

Now you

~~I have put a challenge on 46.101.80.6 on ports 10000 to 10004. The goal is to make the program spit out the flag by using an already free()'d variable. Try it out!~~

The challenge is not up anymore. You can still try with the code on github. Do not hesitate on sending me any further questions you might have.

~~The flag is in the form: **SP{contents_of_the_flag}**~~

Hints

– There is one chunk already allocated at the beginning!

– Bear in mind that all addresses are 64bit. This means 8 bytes.

– Chunks are **pointers** to a **pointer** to an **array of char** (string)

```
char ** strchunks[MAXCHUNKS];
```

– Yes, there is a double-free case but it is not exploitable

– Yes and no, if you could leak addresses and knew the offset to libc, you could probably get a shell :)

Happy pwning!!!

Further reading

[1] Ode to the use-after-free by Chris Evans @ Google Project Zero

Page Saved!

Add Tags

Get in touch

Please select an area that you would like to enquire about and we'll get back to you as soon as possible.

Select an interest
General

Name

Email Address

Contact Number

Get In Touch

By clicking 'Send' you agree to SensePost's Terms of Service

Pretoria

+27 (0)12 460 0880

London

+44 (0)203 355 7369

info@sensepost.com

© SensePost 2018