Safe-Linking – Eliminating a 20 year-old malloc() exploit primitive

May 21, 2020

Research by: Eyal Itkin

Overview

One of our goals for every research project we work on in Check Point Research is to get an intimate understanding of how software work: What components do they contain? Are they vulnerable? How can attackers exploit these vulnerabilities? And more importantly, how can we protect against such attacks?

In our latest research, we created a security mechanism, called "Safe-Linking", to protect `malloc()`'s single-linked lists from tampering by an attacker. We **successfully** pitched our approach to maintainers of core open-source libraries, and it is now **integrated** into the most common standard library implementation: **glibc** (Linux), and its popular embedded counterpart: **uClibc-NG**.

It is important to note that Safe-Linking is not a magic bullet that will stop all exploit attempts against modern-day heap implementations. However, this is another step in the right direction. From our past experience, this specific mitigation would have blocked several major exploits that we demonstrated throughout the years, thus turning "broken" software products to ones that are "unexploitable."

Background

From the early days of binary exploitation, the heap internal data structures have been a prime target for attackers. By understanding the way the heap's `malloc()` and `free()` work, attackers were able to leverage an initial vulnerability in a heap buffer, such as a linear buffer overflow, into a stronger exploit primitive such as an Arbitrary-Write.

An example of this is detailed in the Phrack article from 2001: Vudo Malloc Tricks (https://phrack.org/issues/57/8.html). This article explores the internals of multiple heap implementations, and describes what is now called "Unsafe-Unlinking." Attackers that modify the `FD` and `BK` pointers of double-linked lists (such as the Small-Bins, for example) can use the `unlink()` operation to trigger an Arbitrary-Write, and thus achieve code execution over the target program.

And indeed, glibc version 2.3.6 from 2005 embedded a fix to this known exploit primitive called "Safe-Unlinking." This elegant fix verifies the integrity of the double-linked node before unlinking it from the list, as can be seen in Figure 1:

```
if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  malloc_printerr ("corrupted double-linked list");
```

Figure 1: Safe-Unlinking on action – checking that `p->fd->bk == p`.

While this exploit primitive was blocked more than 15 years ago, at the time no one came up with a similar solution to protect the pointers of the single-linked lists. Leveraging this weak spot, attackers shifted their focus to these unprotected single-linked lists such as Fast-Bins and also TCache (Thread Cache). Corrupting the single linked-list allows the attacker to gain an Arbitrary-Malloc primitive, i.e. a small controlled allocation in an arbitrary memory address.

In this article, we close this almost 20 year old security gap and show how we created a security mechanism to protect single-linked lists.

Before we dive into the design of Safe-Linking, let's review a sample exploitation that targets the vulnerable Fast-Bins. Readers with a lot of experience in such exploitation techniques should feel free to skip ahead to the section, **Introducing Safe-Linking**.

Fast-Bin Exploitation Test Case – CVE-2020-6007:

During our research on hacking smart lightbulbs (https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/), we found a heap-based buffer overflow vulnerability: CVE-2020-6007 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6007). Through the exploitation process of this high-severity vulnerability, we illustrate how attackers leverage the Fast-Bins' unprotected single-linked lists to transform a heap-based linear buffer overflow into a much more powerful Arbitrary-Write.

In our test case, the heap is a simple dlmalloc (https://gee.cs.oswego.edu/dl/html/malloc.html) implementation, or more specifically, the "malloc-standard" implementation of uClibc (https://www.uclibc.org/) (micro LibC), compiled to a 32bit version. Figure 2 shows the meta-data used by this heap implementation:

```
struct malloc_chunk {

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

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

Figure 2: The `malloc_chunk` structure as used in the heap's implementation.

Notes:

- When a buffer is allocated and used, the first two fields are stored **before** the user's data buffer.
- When the buffer is freed and placed in a Fast-Bin, the third field is also used and points to the next node in the Fast-Bin's linked list. This field is located at the first 4 bytes of the user's buffer.
- When the buffer is freed and isn't placed in a Fast-Bin, both the third and fourth fields are used as part of the doubly-linked list. These fields are located at the first 8 bytes of the user's buffer.

The Fast-Bins are an array of various sized "bins", each holding a single-linked list of chunks up to a specific size. The minimal bin size contains buffers up to 0x10 bytes, the next holds buffers of 0x11 to 0x18 bytes, and so on.

Overflow plan

Without diving into specifics, our vulnerability gives us a small, yet controllable, heap-based buffer overflow. Our master plan is to overflow an adjacent free buffer that is located in a Fast-Bin. Figure 3 shows how the buffers look **before** our overflow:

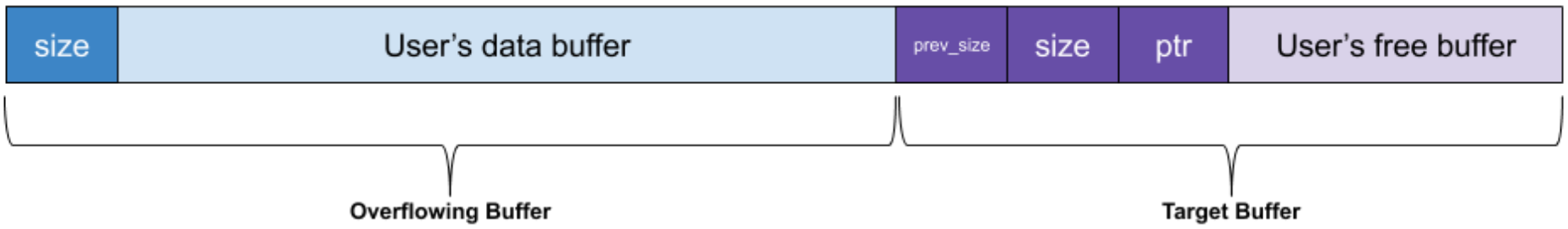

Figure 3: Our controlled buffer (in blue) placed before a freed buffer (in purple).

Figure 4 shows the same two buffers **after** our overflow:

Figure 4: Our overflow modified the `size` and `ptr` fields of the freed buffer (shown in red).

Using our overflow, we modify what we hope is the single-linked pointer of a Fast-Bin record. By changing this pointer to our own arbitrary address, we can trigger the heap into thinking that a new freed chunk is now stored there. Figure 5 shows the Fast-Bin's corrupt single-linked list, as it appears to the heap:

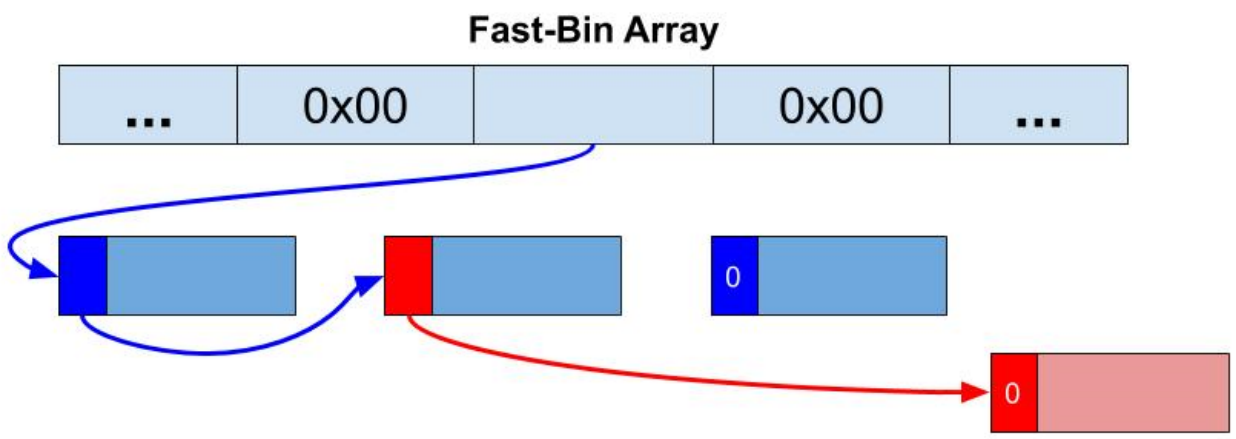

Figure 5: The Fast-Bin's corrupt single-linked list in red.

By triggering a sequence of allocations of the size which matches that of the relevant Fast-Bin, we gain a Malloc-Where primitive. The rest of the technical details on constructing a full code execution exploit, as well as the lightbulb research in full, can be found in our blog post (https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/).

Note: Some of you might say that the Malloc-Where primitive we gained is constrained, as the dummy "Free Chunk" should begin with a size field that matches that of the current Fast-Bin. However, this additional verification check was only implemented in glibc, and is missing from uClibc-NG. Therefore, we have no limitations whatsoever for our Malloc-Where primitive.

Introducing Safe-Linking:

After finishing the lightbulbs research, I had some time to spare before 36C3 began, and my plan was to solve some pwn challenges from recent CTF competitions. Instead, I found myself thinking again about the recently developed exploit. I've been exploiting heap-based buffer overflows the same way for almost a decade, always targeting a single-linked list in the heap. Even in the CTF challenges, I still focused on the vulnerable single-linked lists of the TCache. Surely there is some way to mitigate this popular exploit primitive.

And this is how the concept of Safe-Linking arose. Safe-Linking makes use of randomness from the Address Space Layout Randomization (ASLR), now heavily deployed in most modern operating systems, to "sign" the list's pointers. When combined with chunk alignment integrity checks, this new technique protects the pointers from hijacking attempts.

Our solution protects against 3 common attacks regularly used in modern day exploits:

- **Partial pointer override:** Modifying the lower bytes of the pointer (Little Endian).
- **Full pointer override:** Hijacking the pointer to a chosen arbitrary location.

- **Unaligned chunks:** Pointing the list to an unaligned address.

Threat Modeling

In our threat model, the attacker has the following capabilities:

- Controlled linear buffer overflow / underflow over a heap buffer.
- Relative Arbitrary-Write over a heap buffer.

It's important to note that our attacker doesn't know where the heap is located, as the heap's base address is randomized together with the `mmap_base` by the ASLR (more on this topic in the next section).

Our solution raises the bar and blocks our attacker's heap-based exploit attempts. Once deployed, attackers must have an additional capability in the form of heap-leak / pointer-leak. An example scenario for our protection is a position-dependent binary (loads without ASLR) that has a heap overflow when parsing incoming user input. This was exactly the case in the example we've shown before in the lightbulb research.

Until now, an attacker was able to exploit such targets without any heap leak, and with only minimal control over the heap's allocations, by depending solely on the fixed addresses of the binary. We can block such exploit attempts, and leverage the heap's ASLR to gain randomness when we redirect heap allocations to fixed addresses in the target binaries.

Protection

On Linux machines, the heap is randomized via `mmap_base` which follows the following logic:

```
1.          random_base = ((1 << rndbits) - 1) << PAGE_SHIFT)
```

`rndbit` defaults to 8 on 32 bit Linux machines, and 28 on 64 bit machines.

We denote the address in which the single-linked list pointer is stored as `L`. We now define the following calculation:

$$Mask := (L >> PAGE\_SHIFT)$$

According to the ASLR formula shown above, the shift positions the first random bits from the memory address right at the LSBit of the mask.

This leads us to our protection scheme. We denote the single-linked list pointer with `P` and this is how the scheme looks:

- `PROTECT(P) := (L >> PAGE_SHIFT) XOR (P)`
- `*L = PROTECT(P)`

The code version:

```
1.  #define PROTECT_PTR(pos, ptr, type)  \
2.          ((type)((((size_t)pos) >> PAGE_SHIFT) ^ ((size_t)ptr)))
3.
4.  #define REVEAL_PTR(pos, ptr, type)   \
5.          PROTECT_PTR(pos, ptr, type)
```

This way, the random bits from the address `L` are placed on top of the LSB of the stored protected pointer, as can be seen in Figure 6:

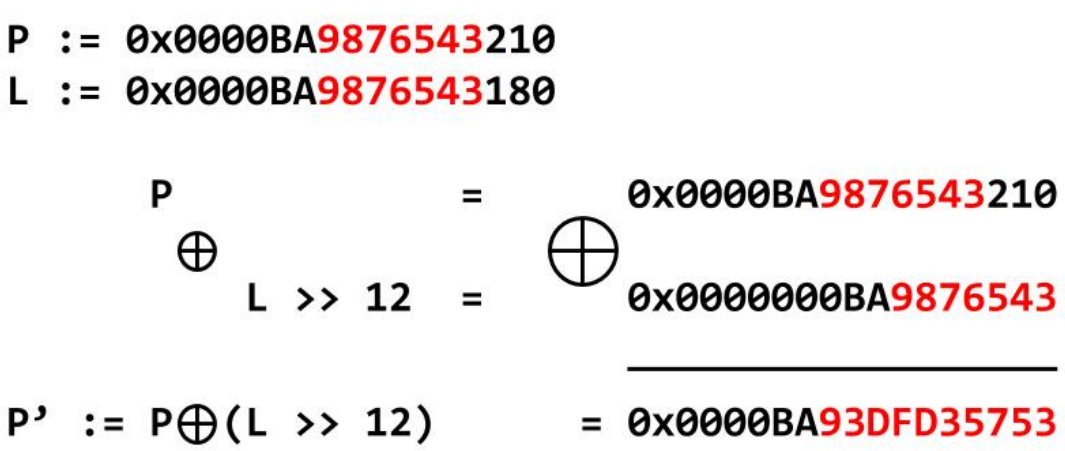

Figure 6: The masked pointer `P'` is covered with random bits, as shown in red.

This protection layer prevents an attacker without the knowledge of the random ASLR bits (shown in red) from modifying the pointer to a controlled value.

However, if you paid attention, you can easily see that we are at a disadvantage against the Safe-Unlinking mechanism. While the attacker can't properly hijack the pointer, we are also limited as we can't check if a pointer modification occurred. This is where an additional check takes place.

All allocated chunks in the heap are aligned to a known fixed offset which is usually 8 bytes on 32 bit machines, and 16 on 64 bit machines. By checking that each `reveal()`ed pointer is aligned accordingly, we add two important layers:

- Attackers must correctly guess the alignment bits.
- Attackers can't point chunks to unaligned memory addresses.

On 64 bit machines, this statistical protection causes an attack attempt to fail 15 out of 16 times. If we go back to Figure 6, we can see that the value of the protected pointer ends with the nibble 0x3, meaning that an attacker must use the value 0x3 in his overflow, or else he will corrupt the value and fail the alignment check.

Even on its own, this alignment check prevents known exploit primitives, such as the one described in this article (https://quentinmeffre.fr/exploit/heap/2018/11/02/fastbin_attack.html) that describes how to point the Fast-Bin at the `malloc()` hook to immediately gain code execution.

Side Note: On intel CPUs, glibc still uses an alignment of 0x10 bytes on both 32 bits **and** 64 bits architectures, unlike the common case we just described above. This means that for glibc, we provide enhanced protection on 32 bits, and can statistically block 15 out of 16 attacks attempts.

Example Implementation

Figure 7 shows a snippet from the initial patch that we've submitted to glibc:

```
@@ -4196,11 +4226,15 @@ _int_free (mstate av, mchunkptr p, int have_lock)
       LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
       for (tmp = tcache->entries[tc_idx];
        tmp;
-       tmp = tmp->next)
+       tmp = REVEAL_PTR (tmp->next))
+       {
+        if (__glibc_unlikely (!aligned_OK (tmp)))
+       malloc_printerr ("free(): unaligned chunk detected in tcache 2");
         if (tmp == e)
       malloc_printerr ("free(): double free detected in tcache 2");
       /* If we get here, it was a coincidence.  We've wasted a
          few cycles, but don't abort.  */
+       }
     }

   if (tcache->counts[tc_idx] < mp_.tcache_count)
@@ -4264,7 +4298,7 @@ _int_free (mstate av, mchunkptr p, int have_lock)
      add (i.e., double free).  */
   if (__builtin_expect (old == p, 0))
     malloc_printerr ("double free or corruption (fasttop)");
-  p->fd = old;
+  p->fd = PROTECT_PTR (&p->fd, old);
   *fb = p;
```

Figure 7: Example code snippet from the initial version of our patch, as sent to glibc.

While the patch was since cleaned, we can still see that the required code modification for protecting glibc's TCache is small and simple. Which brings us to the next section, benchmarking.

Benchmarking

Benchmarking showed that the added code sums up to 2-3 asm instructions for `free()` and 3-4 asm instructions for `malloc()`. On glibc, the change was negligible and wasn't measurable even when summing up **1 billion** (!) `malloc()`/`free()` operations on a single vCPU in GCP. The tests were run on a 64 bit version of the library. Here are the results after running glibc's benchmarking test `malloc-simple` on buffers of size 128 (0x80) bytes on the same GCP server:

	Vanilla Version	Using Safe-Linking
max_rss	1676	1672
main_arena_st_allocs_0025_time	**77.511**	81.607
main_arena_st_allocs_0100_time	94.439	**94.035**
main_arena_st_allocs_0400_time	**93.528**	97.111
main_arena_st_allocs_1600_time	161.315	**159.065**
main_arena_mt_allocs_0025_time	**117.634**	123.422
main_arena_mt_allocs_0100_time	**144.564**	146.074
main_arena_mt_allocs_0400_time	159.591	**151.536**
main_arena_mt_allocs_1600_time	230.882	**215.798**
thread_arena__allocs_0025_time	120.277	**116.577**
thread_arena__allocs_0100_time	146.871	**142.024**
thread_arena__allocs_0400_time	**157.205**	158.380
thread_arena__allocs_1600_time	**223.460**	233.216

Figure 8: Benchmarking results for glibc's `malloc-simple` test.

The faster results for each test are marked in **bold.** As we can see, the results are pretty much the same, and in half of the tests the patched version was **faster**, which doesn't really make sense. This usually means that the noise level on the server is way higher than the actual impact of the added feature on the overall results. Or in short, it means that the added overhead from our new feature is negligible, and that's good news.

As another example, the worst impact on tcmalloc's (gperftools (https://github.com/gperftools/gperftools)) benchmarking tests was an overhead of 1.5%, while the average was only 0.02%.

These benchmarking results are due to the slim design of the proposed mechanism:

- The protection has no memory overhead.
- The protection needs no initialization.
- No special source of randomness is needed.
- The protection uses only `L` and `P` which are both present at the time the pointer needs to be `protect()`ed or `reveal()`ed.

It is important to note, and is detailed in glibc's documentation, that both the Fast-Bins and the TCache use single-linked lists to **hold** data. They only support the put/get API, and no common functionality traverses the entire list and keeps it intact. While such functionality does exist, it is only used for gathering malloc statistics (`mallinfo`), and so the added overhead for accessing the single-linked pointer is negligible.

Revisiting our Thread Model

The alignment check reduces the attack surface and mandates that a Fast-Bin or a TCache chunk points to an aligned memory address. This directly blocks known exploit variants, as mentioned above.

Just like Safe-Unlinking (for double-linked lists), our protection relies on the fact that the attacker doesn't know what legitimate heap pointers look like. In the double-linked list scenario, an attacker that can forge a memory struct, and knows what a valid heap pointer looks like, can successfully forge a valid `FD/BK` pair of pointers that won't trigger an Arbitrary-Write primitive, but allows a chunk at an attacker-controlled address.

In the single-linked list scenario, an attacker **without** a pointer leak won't be able to fully control an overridden pointer, due to the protection layer that relies on the randomness inherited from the deployed ASLR. The proposed `PAGE_SHIFT` places the random bits right over the first bit of the stored pointer. Together with the alignment check, this statistically blocks attackers from changing even the lowest bit / byte (Little Endian) of the stored single-linked pointer.

A Plot Twist – Chromium's MaskPtr()

Our intent was to merge Safe-Linking into the leading open sources that implement each kind of heap that contain single-linked lists. One such implementation is tcmalloc (Thread-Cache Malloc) by Google, which, at the time, was only open-sourced as part of the gperftools (https://github.com/gperftools/gperftools) repository. Before submitting our patch to gperftools, we decided to have a look at Chromium's Git repository, just in case they might use a different version of tcmalloc. Which, as it turns out, they did.

We will skip the history lesson, and go straight to the results:

- gperftools has issued public releases since 2007, and the latest version is 2.7.90.
- Chromium's tcmalloc looks as if it is based on gperftool's 2.0 version.
- In February 2020, after we already submitted patches to all of the open sources, Google released an official TCMalloc GitHub repository (https://github.com/google/tcmalloc), which differs from both prior implementations.

While examining Chromium's version, we saw that not only is their TCache now based on double-linked lists (now called `FL`, for Free List) and not a single-linked list (originally called `SLL`), they also added a peculiar function called `MaskPtr()`. A closer look at their issue (https://chromiumcodereview.appspot.com/10944024) from 2012 showed the following code snippet:

```
1.  inline void* MaskPtr(void* p) {
2.    // Maximize ASLR entropy and guarantee the result is an invalid address.
3.    const uintptr_t mask =
4.        ~(reinterpret_cast<uintptr_t>(TCMalloc_SystemAlloc) >> 13);
5.    return reinterpret_cast<void*>(reinterpret_cast<uintptr_t>(p) ^ mask);
6.  }
```

The code is remarkably similar to our `PROTECT_PTR` implementation. In addition, the authors of this patch specifically mentioned that, "The goal here is to prevent freelist spraying in exploits."

It looks like Chromium's security team introduced their own version of Safe-Linking to Chromium's tcmalloc, and they did it 8 years ago, which is quite impressive.

From examining their code, we can see that their mask is based on a (random valued) pointer from the code section (`TCMalloc_SystemAlloc`), and not a heap location as is used in our implementation. In addition, they shift the address by the hard coded value 13, and also invert the bits of their mask. As we failed to find the documentation for their design choices, we can read from the code that the bit inversion is used to guarantee the result is an invalid address.

From reading their logs, we also learned that they estimated the performance overhead of this feature as less than 2%.

Compared to our design, Chromium's implementation implies an additional memory reference (to the code function) and an additional asm instruction for the bit flipping. Not only that, their pointer masking is used without the additional alignment check, and therefore the code can't catch a pointer modification in advance without crashing the process.

Integration

We implemented and tested the patches for successfully integrating the proposed mitigation to the latest versions of glibc (`ptmalloc`), uClibc-NG (`dlmalloc`), gperftools (`tcmalloc`) and later, Google's brand-new TCMalloc. In addition, we also pointed Chromium's development team to our version of Safe-Linking as was submitted to gperftools, in the hope that some of our gperftools-specific performance boosts will find their way into Chromium's version.

When we started working on Safe-Linking, we believed that integrating Safe-Linking to these 3 (now 4) dominant libraries will lead to a wider adoption by other libraries, both in the open source community and in closed source software in the industry. The fact that a basic version of Safe-Linking was already embedded into Chromium since 2012 proves the maturity of this solution.

Here are the integration results, as they are at the time of writing this blog post.

glibc (ptmalloc)

Status: Integrated. Will be released in version 2.32 in August 2020.
Activation: On by default.
The maintainers of GNU's glibc project were very cooperative and responsive. The main hurdle was signing the legal documents that will allow us, as employees in a corporation, to donate GPL-licensed source code to GNU's repository. Once we overcame this issue, the process was really smooth, and the latest version of our patch was committed (https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=a1a486d70ebcc47a686ff5846875eacad0940e41) to the library, rea
be included in the upcoming version.

We would like to thank the maintainers of glibc for their cooperation throughout this entire process. Their willingness to integrate an "on by default" security feature into their project was heart-warming, especially when compared to our initial expectations, and to the response we received from other repositories.

uClibc-NG (dlmalloc)

Status: Released in version v1.0.33 (https://gogs.waldemar-brodkorb.de/oss/uclibc-ng/src/v1.0.33).
Activation: On by default.
Submitting our feature to uClibc-NG was very easy, and it was integrated right away in this commit (https://gogs.waldemar-brodkorb.de/oss/uclibc-ng/commit/886878b22424d6f95bcdeee55ada72049d21547c). We can proudly say that Safe-Linking is already released as part of uClibc-NG version v1.0.33 (https://gogs.waldemar-brodkorb.de/oss/uclibc-ng/src/v1.0.33). If we go back to our research on smart lightbulbs, this feature would have blocked our exploit and would have forced us to find an additional vulnerability in the product.

Once again, we want to thank the maintainers of uClibc-NG for their cooperation in this process.

gperftools (tcmalloc)

Status: Undergoing integration.
Activation: Off by default.
Although we earlier mentioned Chromium's `MaskPtr()` functionality, available since 2012, this feature didn't find its way to either of the public versions of tcmalloc. And so, we tried our luck in submitting Safe-Linking to gperftools's implementation of tcmalloc.

Due to the complex status of the gperftools repository, now that Google's official TCMalloc repository is public, this process was making progress, but slowly. In this pull request (https://github.com/gperftools/gperftools/pull/1161) from the beginning of January 2020, you can see our efforts to integrate this feature into the repository. The initial response was the one we dreaded: our feature "ruins performance." As a reminder, when using the repository's benchmarking suite, the worst case result was an overhead of 1.5%, and the average was only 0.02%.

Eventually, with some reluctance, we settled on adding this feature as "off by default", hoping that someday someone will activate this feature on its own. This feature wasn't merged yet, but we hope that it will be in the near future.

We would still like to thank the sole maintainer of this project, who offered to do all of the necessary plumbing so as to allow users the configuration option to enable Safe-Linking.

TCMalloc (tcmalloc)

Status: Rejected.
Activation: N/A.
We submitted our patch to TCMalloc in this pull request (https://github.com/google/tcmalloc/pull/19), which, sadly, was rejected right away. Once again, we heard, "The performance costs of this are too high to merge", and that they are not going to integrate it even as an "off by default" configurable feature: "While macro-protected, it adds another configuration that needs to be built and regularly tested to ensure everything stays working." We couldn't find any representative from Google that would help us resolve this conflict with the maintainers of the repository, and we left it as is.

Unfortunately, it looks like Google's most common implementation of `malloc()`, which is used in most of their C/C++ projects (as mentioned in TCMalloc's documentation), wasn't going to integrate a security feature that will make it harder to exploit vulnerabilities in their projects.

Final Note

Safe-Linking is not a magic bullet that will stop all exploit attempts against modern-day heap implementations. However, this is another step in the right direction. By forcing an attacker to have a pointer leak vulnerability before he can even start his heap-based exploit, we gradually raise the bar.

From our past experience, this specific mitigation would have blocked several major exploits that we implemented through the years, thus turning "broken" software products to "unexploitable" (at least with the vulnerabilities we had at the time of the respective research projects).

It is also important to note that our solution isn't limited only to heap implementations. It also enables endangered data structures, with single-linked list pointers that are located near user buffers, to get integrity protection without any addit memory overhead, and with a negligible run-time impact. The solution is easily extendable to every single-linked list in a

system with ASLR.

It was a tough journey, starting from when we first tackled this issue at the end of 2019, to designing a security mitigation, and finally integrating it as a security feature that is available by default in two of the world's most prominent libc implementations: glibc and uClibc-NG. In the end, we had much better results in integrating this feature than we initially expected, and so we wish to once again thank all of the maintainers and researchers that helped make this idea a reality. Gradually, step-by-step, we raise the exploitation bar, and help protect users all over the world.

RELATED ARTICLES

ce

2022 Security Report: Software Vendors saw 146% Increase in Cyber Attacks in 2021, marking Largest Year-on-Year

New Wormable Android Malware Spreads by Creating Auto-Replies to Messages in WhatsApp

Vulnerability in Google Play Core Library Remains Unpatched in Google Play Applications

Graphol
exploits
fingerpri

PUBLICATIONS

GLOBAL CYBER ATTACK REPORTS (HTTPS://RESEARCH.CHECKPOINT.COM/CATEGORY/THREAT-INTELLIGENCE-REPORTS/)

RESEARCH PUBLICATIONS (HTTPS://RESEARCH.CHECKPOINT.COM/CATEGORY/THREAT-RESEARCH/)

IPS ADVISORIES (HTTPS://WWW.CHECKPOINT.COM/ADVISORIES/)

CHECK POINT BLOG (HTTP://BLOG.CHECKPOINT.COM/)

DEMOS (HTTPS://RESEARCH.CHECKPOINT.COM/CATEGORY/DEMOS/)

TOOLS

SANDBLAST FILE ANALYSIS (HTTPS://THREATEMULATION.CHECKPOINT.COM/)

URL CATEGORIZATION (HTTPS://WWW.CHECKPOINT.COM/URLCAT/)

INSTANT SECURITY ASSESSMENT (HTTP://WWW.CPCHECKME.COM/CHECKME/)

LIVE THREAT MAP (HTTPS://THREATMAP.CHECKPOINT.COM/THREATPORTAL/LIVEMAP.HTML)

ABOUT US (HTTPS://RESEARCH.CHECKPOINT.COM/ABOUT-US/) CONTACT US (HTTPS://RESEARCH.CHECKPOINT.COM/CONTACT/)

SUBSCRIBE (HTTPS://RESEARCH.CHECKPOINT.COM/SUBSCRIPTION/)