

Re: [Vuln-Dev Challenge] – VulnDev1.c Summary

Source: <http://www.derkeiler.com/Mailing-Lists/securityfocus/vuln-dev/2003-05/0091.html>

From: Jason Royes (*jroyes_at_da-experts.com*)

Date: 05/21/03

To: Aaron Adams <aadams@securityfocus.com>

Date: 20 May 2003 22:14:33 -0400

This may be totally incorrect, but here's my armchair analysis.

On glibc systems: 0x4, 0x5 and 0x8 all have the IS_MMAPPED bit turned on (0x9 does not), meaning the chunk is allocated with mmap. Which means they're free'd by munmap_chunk (when free(buf2) is called), which in turn calls munmap on an invalid memory address. The invalid memory address is equal to p - p->prev_size, where prev_size is the last 4 bytes of buf1 (i.e. some big number). This results in a segfault.

— snip:malloc.c —

```
static void
internal_function
#ifdef __STD_C
munmap_chunk(mchunkptr p)
#else
munmap_chunk(p) mchunkptr p;
#endif
{
/* JR: size = 0x10{4,5,8} */
INTERNAL_SIZE_T size = chunksize(p);
int ret;

assert(chunk_is_mmapped(p));
assert(!((char*)p >= sbrk_base && (char*)p < sbrk_base +
sbrked_mem));
assert((n_mmaps > 0));
/*
* JR: ensure length is multiple of pagesize
* prev_size = < last 4 bytes of buf1 >
*/
assert(((p->prev_size + size) & (malloc_getpagesize-1)) == 0);

n_mmaps--;
mmaped_mem -= (size + p->prev_size);

/* unmap */
/* JR: Memory Access Violation p->prev_size is huge */
ret = munmap((char *)p - p->prev_size, size + p->prev_size);
```

SecurityFocus Vuln-Dev: Re: [Vuln-Dev Challenge] – VulnDev1.c Summary

```
/* munmap returns non-zero on failure */  
assert(ret == 0);  
}
```

```
--  
EOF
```

On Tue, 2003-05-20 at 19:19, Aaron Adams wrote:

```
> For each challenge program that we release we will do our best to post a  
> summary that includes our intentions for releasing the particular program.  
> We will also explain and summarize some of the findings by people who took  
> part in the discussion.
```

```
>
```

```
> VulnDev2.c is currently in the works and will hopefully be released May  
> 21st.
```

```
>
```

```
> VulnDev1.c Summary
```

```
> -----
```

```
>
```

```
> -- Summary --
```

```
>
```

```
> The VulnDev1.c challenge program was written to encourage people to look  
> into the internal workings of heap allocation on various systems. It was  
> designed in hopes that readers would research into why an off-by-one in  
> the heap is exploitable under some circumstances.
```

```
>
```

```
> It was specifically written to be exploited on a system implementing the  
> Doug Lea Malloc implementation [1]. This includes the Linux and GNU/HURD  
> operating systems, and possibly others. As a result, this issue had  
> varying results depending on the type of system under which it was  
> invoked. Some participants noted a segmentation fault would not occur on  
> AIX and Windows systems. This is due to differing allocation algorithms  
> and memory management features. *BSD systems are also unaffected by this  
> issue as the algorithm used (PHK) does not implement inline memory  
> management. Meaning that information used for keeping track and managing  
> the status of the heap is not corruptable by overrunning a buffer in the  
> heap.
```

```
>
```

```
> I assume that the reader of this summary is familiar with the internals of  
> the Doug Lea algorithm as well has heap-based exploitation methods [2, 3].  
> I will not be delving into the internals of these as they have all been  
> well documented.
```

```
>
```

```
> -- Details --
```

```
>
```

```
> VulnDev1.c contained a simple flaw within the for() loop.
```

```
>
```

```
> for (i = 0; i <= SIZE && i != '\0'; i++)  
>     buf1[i] = p[i];
```

```
>
```

```
> This flaw allowed a user invoking the program to overwrite 1 byte of heap  
> memory adjacent to memory allocated for buf1. Due to the Doug Lea  
> implementation, this 1 byte of memory corruption can be leveraged to  
> execute our own instructions. Two contributors to the discussion  
> demonstrated this successfully.
```

```
>
```

```
> The one key attribute of VulnDev1.c which made exploitation possible was  
> the following declaration:
```

```
>
```

```
> #define SIZE 252
```

```
>
```

```
> As some posts indicated, simple modifications of this value would prevent
```

SecurityFocus Vuln-Dev: Re: [Vuln-Dev Challenge] – VulnDev1.c Summary

> a segmentation fault from occurring when SIZE is overrun. This is due to
> the behavior of the malloc() function when allocating buffers of various
> sizes.
>
> As described in Once Upon A free(), each chunk size passed to
> malloc() has a minimum of 4 bytes additional overhead. This is to
> accommodate for the [SIZE] field of the adjacent chunk header. Also, due
> to the 3 least significant bits of each [SIZE] value being used as flags,
> the [SIZE] value of each chunk is padded to the next 8 byte boundary.
>
> The following examples depict this behavior:
>
> malloc(256) [SIZE = 264]
> malloc(15) [SIZE = 24]
> malloc(0) [SIZE = 8]
>
> And most importantly:
>
> malloc(252) [SIZE = 256]
>
> As shown above, depending on the [SIZE] padding and the subsequent layout
> of the chunk within memory, the last byte of a user-controllable buffer
> may be directly adjacent to the first byte of the next headers [SIZE]
> value. This condition will only occur when the size of an allocated buffer
> + 4 falls on an 8 byte boundary, such as 252, 12, 1020, etc. In
> situations where this does not occur, overwriting 1 byte of memory will
> result in the corruption of a padding byte. However, in situations similar
> to the memory layout caused by VulnDev1.c, overwriting 1 byte of memory
> will allow for the corruption of the LSB of the adjacent [SIZE] field on
> little endian machines. A variety of posts touched on this issue when
> slightly modifying the SIZE variable used by malloc().
>
> Now that we know that this bug can be used to overwrite a sensitive value
> in memory, the question is whether or not this can be leveraged to execute
> our own instructions. As was shown, this is indeed possible and I will
> explain why. Just as a note, due to the layout of the program, at first
> glance it may appear that exploitation could occur during free(buf2). In
> actuality, exploitation could occur during free(buf1). There are also some
> anomalies with exploitation which I will do my best to address later on.
>
> When buf2 is allocated (buf2 = malloc(SIZE)) the additional 4 bytes are
> added to SIZE, resulting in a [SIZE] value of 0x0100 (256). Because buf1
> has also been allocated, and thus is in use, the PREV_INUSE flag will also
> be set, causing the [SIZE] to be 0x0101.
>
> When the 1 byte of memory corruption occurs the LSB of buf2's [SIZE] will
> be overwritten. For example's sake we will use the value 0xC. This will
> result in buf2's [SIZE] value being 0x010C. This corrupted value is first
> referenced during the call to free(buf1) and this is where exploitation
> takes place.
>
> At some point during execution the free(buf1) call checks whether the
> adjacent forward or backward chunks are free. If so, the chunks will be
> coalesced to avoid contiguous free chunks. The [SIZE] value of buf1's
> header is first checked for the PREV_INUSE flag to see if buf1 and the
> previous chunk should be coalesced. This is followed by a check to see if
> the forward chunk, in our case buf2, is free. To make the check the
> PREV_INUSE flag AFTER buf2 must be tested. This is accomplished by
> referencing the address at *buf1 + ([SIZE] field of buf1) + ([SIZE] field
> of buf2). This third chunk's [SIZE] value is then referenced via this
> location. Under normal circumstances this would place the test at the
> first bytes of data in the chunk following buf2. In the context of

SecurityFocus Vuln-Dev: Re: [Vuln-Dev Challenge] – VulnDev1.c Summary

```
> exploitation of VulnDev1.c however the buf2[SIZE] value has been
> corrupted. This will result in the check for PREV_INUSE flag occurring
> further in the heap, beyond the chunk following buf2. In our case, if
> buf2[SIZE] is 0x010C, then the check will occur at the offset *buf1 + 524.
> In our situation this offset will place us in unused heap memory which has
> been initialized to zero.
>
> This will result in the PREV_INUSE flag appearing unset, making free()
> believe that buf2 is in fact free. Forward consolidation will then occur
> and the unlink() function will be called on buf2. This inturn can be used
> to our advantage by populating the first 8 bytes of buf2 with fake [FD]
> and [BK] pointers, which when referenced by unlink() will be corrupted.
>
> As shown by the two exploits released, this can lead to exploitation by
> overwriting the free GOT entry. Other values which could be overwritten
> include .dtors or a function pointer within free() itself. It should be
> noted that if the second free() is allowed to be called and the corrupted
> [SIZE] is again referenced, varying functionality may occur.
>
> That's all. Overwriting the free GOT entry to point to our own
> instructions somewhere in memory will allow us to execute our own code,
> theoretically with elevated privileges of course. Off-by-one bugs in the
> heap have been found in the wild and I hope the list's discussion and
> summary will help people understand how exploitation works.
>
> -- Additional Discussion --
>
> When I first wrote the program I considered a scenario, dependent on bytes
> chosen for the 1 byte overwrite, that would result in the program
> exiting cleanly. I also believed that in some situations exploitation
> could occur during the second free(), however I was unable to reproduce
> the situation and have since determined that exploitation during the
> second free() is likely not possible.
>
> Although I was unable to reproduce my hypothesis, Matrix and Marco Ivaldi
> described a situation that was similar to the scenario I had anticipated.
> No discussion really followed their findings so I will do my best to shed
> a little light on the issues. It should be noted that there is some
> behavior that I have not been able to figure out. Any corrections or
> additional information regarding these observations is much appreciated
> and anticipated.
>
> Matrix:
>
> > It's true that the exploit will work on the first free() for any value
> > of the overflow char > 0x9, however I noticed some different things
> > happen for values < 0x9. If the overflow char is 0x1, 0x2, 0x6, 0x7, or
> > 0x9 the program will exit cleanly without a segfault. If the overflow
> > char is 0x4 or 0x5, the first free() will execute find, and the program
> > will segfault on the second free(). I don't know if these cases are
> > exploitable or not. And finally, if the overflow char is 0x8 the program
> > segfaults in the first free() (like when the char is > 0x9)
>
> If the corrupted size value is small enough (0x1, 0x2, 0x6, 0x7) the
> value will be ignored during forward consolidation. This is again due to
> the first 3 bits of the value being used as flags. All 4 of these values
> will consume only the 3 least significant bits of the [SIZE] field. As a
> result, the correct [SIZE] (256) will be used for the forward
> consolidation check. As buf2 will correctly be shown as INUSE, unlink()
> will not be called. When the call to free(buf1) is completed, the
> PREV_INUSE flag of buf2 will appropriately be removed and the [PREV_SIZE]
> will be updated accordingly. As a result, exploitation during these cases
```

SecurityFocus Vuln-Dev: Re: [Vuln-Dev Challenge] – VulnDev1.c Summary

```
> is not possible.
>
> I have not been able to figure out why 0x4 and 0x5 will trigger a segfault
> during the second free() as I would expect their behavior to mimic those
> values above. Especially since they also only consume the 3 least
> significant bits of buf2 [SIZE]. Any takers?
>
> The first free() segfaults with a value of 0x8 because it affects the 4th
> bit of buf2 [SIZE] thus affecting the forward consolidation check. This
> should put it ahead in heap memory and result in an unset PREV_INUSE flag.
>
> As for 0x9, again I would expect the behavior to mimic that of 0x8. I'm
> not sure why 0x9 does not behave the same way.
>
> Any questions or corrections to the summary, please feel free to post to
> the list as it will benefit everyone. If necessary, I will participate in
> the list discussion.
>
> -- References --
>
> [1] Doug Lea Malloc Implementation
> http://gee.cs.oswego.edu/dl/html/malloc.html
>
> [2] Once Upon A free()
> http://www.phrack.org/phrack/57/p57-0x09
>
> [3] Vudo - An object superstitiously believed to embody magical powers
> http://www.phrack.org/phrack/57/p57-0x08
>
> EOF
>
> Aaron Adams
--
Jason Royes
Data Access Experts
```