Annales UMCS Informatica AI XI, 4 (2011) 1–10
DOI: 10.2478/v10065-011-0030-7

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Cryptographic software: vulnerabilities in implementations

Michał Łuczaj[1]*

[1] *Institute of Telecommunications, Warsaw University of Technology*
Poland

Abstract – Security and cryptographic applications or libraries, just as any other generic software products may be affected by flaws introduced during the implementation process. No matter how much scrutiny security protocols have undergone, it is — as always — the weakest link that holds everything together to makes products secure. In this paper I take a closer look at problems *usually* resulting from a simple human made mistakes, misunderstanding of algorithm details or a plain lack of experience with tools and environment. In other words: everything that can and will happen during software development but in the fragile context of cryptography.

1 Introduction

I begin with a brief introduction of typical mistakes and oversights that can be made during program implementation in one of the most popular programming languages, C [**1**]. I also explain the concept of exploitable memory corruption, how critical it is and where it leads from the attacker's point of view.

A set of real-world examples is given. Some well known previously disclosed vulnerabilities are brought to illustrate how a flaw (sometimes even an innocent looking) can fatally injune security of the whole protocol, algorithm. There is much to discuss as failed attempts at implementing cryptographic primitives — or making use of cryptography in general — range broadly. From the operating systems of video game consoles to popular open- and closed- source software packages.

*m.luczaj@elka.pw.edu.pl

2 State of *computing*

For a better understanding of the described issues, it is crucial to review the basic blocks that comprise execution environment of a running application. In this paper, not being an exploit writing tutorial, nor CPU manual, I will stick to the very minimum of system internals and their design concepts. For the sake of simplicity I will also ignore the — more or less subtle — architectural differences of modern computers and assume that we all live happily in 32-bit x86-only land [**2**]. I believe the reader should be aware of all the simplifications I make as they do, in fact, affect how things work in real life.

Although I aim to stay at the high level of abstraction, description of some basic yet important memory segments (more precisely: their functionality) requires us to take a slightly deeper dive into the process address space.

Every instance of a program is provided by operating system kernel with a similar environment where it begins its execution. The kernel, besides fulfilling a standardised role of providing programming interfaces, guarding components (mis)behaviour, managing access to restricted and/or shared resources, etc, is also responsible for initialising and setting up application own workspace [**3**].

Thanks to hardware support of memory management unit, the kernel is able to force each user's space application into believing that the whole machine is just for it, i.e. concurrently running executions of different programs (unlike threads) are separated at the highest tier. This means that, for example, one process has no way of accidental modifying another's memory content. This obviously should not be held as a rule of thumb, but any form of inter-process communication (such as shared memory) is beyond the scope of this paper. What is important, though, is that we are not going to be concerned about processes influencing each other's execution state. Our whole discussion about memory corruption and its consequences will be held within limits of a single virtual address space [**4**].

2.1 Code

Clearly the most important part of process memory is where the actual code is stored. Usually it is not a single memory region as most of the programs will, sometimes indirectly, use external libraries also known as dynamic shared objects. Those additional binaries are loaded into address space whenever there is such a need. This, however, is quite moot in the context of our discussion as the kernel (yet again, thanks to hardware support of MMU) makes sure that code segments are immutable by simply terminating process in case of any illegal modification attempt.

This brings us to mostly correct conclusion that if one wants to arbitrarily change the execution flow of application, brazenly trying to modify the code segment is usually not the best way to go.

2.2 Stack

The second important memory region is stack. From the application and programmer's point of view it is generally used for function parameter passing (caller *pushes* function arguments on the stack, callee *pops* them for its use), automatic temporary variables storage (with a life span of a currently executing procedure) and, luckily for attackers, execution flow book-keeping. Basically, every function, right before calling another piece of code or entering some sub-procedure, stores address of the next instruction on the stack. This way CPU at the end of sub-procedure execution will be told where to jump back and carry on with the instruction flow. Thus, not without reason such, stack-stored address is called *return address*.

Stack not only serves for storage in the last-in-first-out manner, but also grows downwards which means that newer allocations will be placed at lower addresses. In particular, local variables of current function will be stored before the return address. Those two characteristics combined with the information from the paragraph above turn out to be critical: local function variables are „intertwined" with return addresses. It means that if an attacker manages to force application to overflow some local temporal buffer, he or she will probably overwrite the return address. And if that happens, execution flow can be altered and bended to the attacker's will.

From the C programmer's standpoint that would probably look like the example given below. No matter how artificial and unrealistic this code snippet appears to be, it represents the problem in full glory. Let us assume that `func()` was called from a higher level block of code with the user-provided input data, `name`. First we see that `strcpy()` copies data into the constant length stack based space. It does not verify if destination buffer is big enough. `tmp` is being overflow, a return address overwritten and right after `printf()` printed out not so important message, CPU returns at memory address designed by a malicious user.

```
void func(char *name)
{
  char tmp[32];

  strcpy(tmp, name);
  printf("%s logged\n", tmp);
}
```

The described method is a classic example of changing program behaviour by modification of the return address through stack bases buffer overflow [**5**]. Surprisingly it iseffective and in many cases still possible to conduct in spite of many years of fighting for security awareness among programmers and development of various exploitation mitigation techniques.

The return address, being as important as fragile, is obviously not the only attack vector for different forms of stack based memory corruptions. Here we could possibly

start another section about frame pointers, overflowing adjacent local variables, characteristics of C null-terminated strings and such, but that would bring nothing more to the moral: security and integrity of data on the stack is important.

2.3 Heap

Another memory segment that is constantly in use by application is heap. It does not have much in common with the previously described stack use case (small, fast allocations of short life span) and thus it is used and managed in a distinct way.

First of all, as heap is supposed to handle also big memory request, it can be dynamically extended. But every time a programmer asks for additional memory chunk via `malloc()` call, it is being handled by *heap manager*. Which, in turn, is usually a part of standard C library, just like code/functions for handling IO operations, data conversion, etc. Then, if heap manager decides, it is running out of memory to manage, the kernel is asked for additional space. It means that a programmer does not usually handle the management by himself — besides requesting and freeing chunks for his ad hoc needs.

Heap manager turns out to be a pretty complicated piece of code by itself [**6**]. Not only does it have to handle requests of varying sizes, take care of memory fragmentation, detect deliberate or accidental anomalies, but also to do it all in an efficient way, not taking much of processor time for its own needs. The same story is with meta-data. To handle those tasks in an optimal way, several layers of custom data structures are used internally. The question is: where are heap manager's (or rather heap chunks') meta-data kept? In most of popular implementations, yet again and fortunately for attackers, the same memory region where the user's data is allocated is explated. One can notice fatal similarity with a stack and return address placement. By overflowing heap based buffer malicious content can replace adjacent chunk's meta-data wrecking havoc. What is a possible next step is not as clear as in the stack corruption case. It depends on a particular heap manager implementation, while those differ greatly.

It should be also clear that sometimes modifying application specific data (i.e. the content of chunk, not its descriptor) is definitely a lower hanging fruit. Analysing and cheating modern allocator mechanisms may be quite a daunting task. During the years after the first heap metadata-based exploit was demonstrated[**7**], more and more sanity and integrity checks were introduced in heap managers.

The generic concept behind memory corruption attacks should be visible by now. It does not matter if the problem is based on stack, heap or any other usable memory region I did not discuss. The aim is to push program into the state that was unexpected by its creator; steal and abuse the logic. Whenever there exists a possibility of changing content of memory even in a semi-controllable fashion, there are high chances that flow of execution can be altered. And altering can be equal to running attacker's provided code.

My short description of basic issues may be misleading. To fully realise the fact how many possible pitfalls software developer can encounter, especially in the environment

of low level languages such a C, I highly recommend the great book on software security audit by Down, McDonald and Schuh [**8**].

3 Vulnerabilities

In the previous section I have laid out some basic foundations of different forms of memory corruptions. Now I propose a short walk through some real-life examples of implementation flaws discovered by various people.

3.1 Debian OpenSSL

This source code mishandling was nominated for Pwnie Awards 2008 [**9**] in a very distinguished category: *Pwnie for Mass 0wnage*.

```
MD_Update(&m, buf, j);
...
/* purify complains */
MD_Update(&m, buf, j);
```

Debian's package maintainer of OpenSSL was having warnings from Valgrind [**10**] concerning use of uninitialised variables in the PRNG code. OpenSSL is, undoubtedly, very popular library used by many open- and closed- source projects so he commented out two lines from above[1] in an attempt to help other developers. Valgrind did not complain anymore and everything worked just fine. It should be noted here that although „fix" was not submitted upstream it was discussed on the openssl-dev mailing list.

What he did not realise was that PRNG was deliberately using uninitialised memory as a form of low entropy source. Obviously there used to be other sources as well, but they were lethally wounded by this simple code modification. As a net result the only variable input used in PRNG seeding was the current process ID. Taking into consideration that the default maximum process ID under Linux is 32,767, all OpenSSL PRNG operations were seeded with a small spectrum of initial values.

It took one year until Luciano Bello [**11**] discovered the flaw and the hell broke loose. All the keys generated on the Debian-based systems within last year needed to be regenerated: OpenSSH authentication, TLS/SSL certificates, TOR, VPN and many others were equally affected. It did not take long until people started creating weak keys sets for transmission eavesdropping, signature forgery, break ins via SSH, etc.

3.2 Nintendo Wii

After about two years from the introduction of Nintendo Wii console, two very curious bugs in the cryptographic subsystem were published [**12**]. It is clear that

[1] http://svn.debian.org/viewsvn/pkg-openssl/openssl/trunk/rand/md_rand.c?r1=140&r2=141&pathrev=141

Nintendo put a lot of effort into securing their game console. Among various ways of keeping users out of device internals or extending hardware functionality, RSA was used for verification of software packages. Wii was meant to simply reject programs that were not signed with the right key. The implementation flaw by itself is a beautiful example of how seemingly correct code breaks everything rendering whole protection useless. And this is not even any form of memory corruption. My guess would be: a lack of experience from the side of developer, a lack of basic understanding how particular API works.

The issue was in the last phase of signature validation. Blob of memory consisting binary SHA-1 hash of user's data was compared against hash from the signature. Unfortunately, `strncmp()` was used for that instead of `memcmp()`. The important difference between those two functions is: `strncmp()` operates on null-terminated strings, while `memcmp()` on „raw" memory. Such C string convention means for `strncmp()` to end comparison whenever byte `0x00` is found — a termination marker. That resulted in a quite peculiar variant of RSA signature verification where we check SHA-1 by comparing two byte arrays only until any of them happens to contain byte `0x00`. Due to that bug it was enough to match hashes to the point where `0x00` was encountered.

As a picture is worth a thousand words, below there are two 160-bit long SHA-1 hashes that would be considered equal by the Wii system software:

```
006caae51286f05943a8f3d5c2b444d0d0317d6e
0054955cd26b8ab25ad3bd5be7063efecce85f68
```

Second vulnerability was less of implementation specific flaw. This time cryptographic logic was flawed: padding check was missing from the process of signature verification.

3.3 MD6 reference implementation

One of SHA-3 candidates [**13**] was MD6. It did not advance to the second round of competition although it was considered a strong, yet somewhat slow candidate. The US National Institute of Standards and Technology required all entrants to provide reference implementation of hash algorithms. As it turned out researchers from Fortify Software managed to find two buffer overflows in the submitted code [**14**].

Both vulnerabilities were a result of single temporary buffer size miscalculation. MD6 implementation used structure `md6_state` with one of the elements defined as:

```
unsigned char hashval[(md6_c/2)*(md6_w/8)];
```

Assuming:

```
#define md6_w      64
#define md6_c      16
```

...we get size of `md6_state.hashval` equal to 64 bytes. Later on, in function `md6_process` we could see:

```
if (z==1)
  { memcpy(st->hashval, C, md6_c*(w/8));
    return MD6_SUCCESS;
  }
```

...which makes `memcpy()` copy 16*(64/8) = 128 bytes into `st->hashval` overwriting the adjacent `md6_state` elements. To be fair, modified part was the first half of buffer holding zero-terminated string representing hexadecimal value of `hashval` making exploitation potential quite small.

It is worth noting that another problem with out-of-buffer write was also found in another round I candidate's implementation: Blender. This time an array of 3 elements was referenced with the index value of 3. Note that in all C-like syntax languages the first element's index number is 0 and the last one is equal to the length minus one.

3.4 SSH

OpenSSH was always considered one of the most important parts of the world wide Internet infrastructure. Its base purpose is simple: allow secure remote machine administration. But just like any other piece of software it has had its own share of implementation flaws discovered.

An integer overflow that became so famous, getting cast in a movie [**15**] was introduced, as it happens from time to time, ironically while trying to fix another issue (protection against some cryptographic attacks on the SSH protocol). It was found by Michał Zalewski [**16**] that at one place in the code 32-bit integer variable which is set to 0x10000 for large input, is assigned to a 16-bit variable, effectively reducing its value to 0. Then, 0 is given as an argument to `malloc()` call, which according to C standard, is a perfectly sane operation. The smallest possible heap chunk is allocated and execution follows. The conclusions from related Bugtraq post:

> By carefully preparing encrypted data, an attacker can point used, accessible memory [...], and then, he will able to alter dword at chosen address [...]. The attacker can alter stack variables, alter malloc structures, etc, and attack later due to improper execution of daemon code. This condition is relatively difficult to exploit, but there are no technical reasons that would make this impossible.

OpenSSH was also affected by a classic integer overflow resulting from multiplication as discovered by IBM Internet Security Systems. Currently this bug serves as an anti-pattern example in the Common Weakness Enumeration dictionary [**17**]:

```
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i > nresp; i++)
   response[i] = packet_get_string(NULL); }
```

One can see above that if value returned by `packet_get_int()` is 0x40000000 (1073741824) and assuming `sizeof(char*)` equals to 4 on 32-bit machine, then the result of multiplication overflows. Thus argument to `xmalloc()` becomes 0 causing `response` to the point at a very small memory chunk. But that does not stop execution causing the subsequent loop iterations to overflow the heap buffer.

As a side note: integer overflows resulting in heap based buffer size miscalculation for such huge numbers are believed to be quite tough and hard to exploit. That is mainly because the attacker, by having to provide high value, at the same time forces loop into many iterations. This, in turn, can lead to a situation when body of a loop hits unreachable memory address (or in this case, non-writable region) before `packet_get_string(NULL)` can be tricked into making some use of already overflown data.

In general though, more complicated and sophisticated program gets, more chances we have to successfully exploit even otherwise unexploitable cases.

Another bug class that could be found in old versions of OpenSSH is *off-by-one* errors. As the name suggests those vulnerabilities are small and innocent looking but if they lead to a form of memory corruption (and usually that is the case), they do have severe consequences. The one found by Joost Pol[**18**] was beautiful in its simplicity: in the channels management code there was the following sanity check:

```
if (id < 0 || id > channels_alloc) {
 log("channel_lookup: %d: bad id", id);
 return NULL;
}
```

Otherwise, if the validity criterion was met, `id` was used as an index value. Later on that index was used in the process of calculating pointer to `id`-th `Channel` structure OpenSSH was currently managing. And then appropriate data stored there would be used in further processing. Knowing that `channels_alloc` represents a number of allocated channels, one can spot a mistake in the code above. The corrected version is:

```
if (id < 0 || id >= channels_alloc) {
 log("channel_lookup: %d: bad id", id);
 return NULL;
}
```

3.5 NSS

Mozilla Network Security Services is a rich set of libraries designed to support cross-platform development of security-enabled client and server applications[2].

In one of its modules responsible for SSLv2 packets parsing, version 3.10 was vulnerable to stack based buffer overflow because of yet another kind of arithmetic problem: integer underflow [**19**]. As it can be seen in the patch provided by Mozilla, they did

[2]http://www.mozilla.org/projects/security/pki/nss/

actually check for buffer overflow condition and then called `memcpy()` wrapper in the following fashion:

```
/* Is the message just way too big? */
if (keySize > SSL_MAX_MASTER_KEY_BYTES) {
 /* bummer */
 ...
}
MEMCPY(mkbuf+ckLen, kk, keySize-ckLen);
```

One thing is lacking here: no care was taken to make sure that `keySize >= ckLen`. The input data that failed to meet that condition caused integer wraparound from the bottom, i.e. instead of the typical (assuming 16-bit integer types):

0xFFFF + 0x0001 = 0x0000

...what we have to deal with is:

0x0001 − 0x0002 = 0xFFFF

Now, calling `memcpy()` with length argument being so big (or negative depending on type declaration) is definitely not a good idea, resulting here in stack based buffer overflow.

4 Conclusions

As one can see there are numerous ways to make subtle mistakes that will cost dearly. Over the short course of introduction to memory corruption and set of real-world examples we have uncovered only some of them, a proverbial tip of an iceberg. There are many other issues remaining: more U2 arithmetic traps, format string bugs, race conditions, use after free, double free, SEH exploitation strategies, etc.

One more class of bugs I would like to write about are those resulting from code re-factoring. Not changing the logic, not moving data from stack to heap and vice versa, not anything serious like that. But simple reformatting. Imaging the following example, where a programmer defined a small buffer and incorporated some sanity check:

```
char my_buffer[128];

if (ulen < sizeof(my_buffer))
  goto bail_out;
```

Later on he had some free time on his hands and decided that it is better to rewrite it in the following way, increasing readability:

```
#define SMALL_BUFF_SIZE 128
char my_buffer[SMALL_BUFF_SIZE];
```

```
if (ulen < sizeof(SMALL_BUFF_SIZE))
  goto bail_out;
```

The obvious issue here is with the `if` condition. Instead of:

```
ulen < sizeof(SMALL_BUFF_SIZE)
```

...there should be:

```
ulen < SMALL_BUFF_SIZE
```

This is because `sizeof(INTEGER_CONSTANT)` will always return 4 making our sanity check moot. It sounds pretty unrealistic to make this kind of mistake, right? Right. And so finding this kind of bug in one of the mentioned software packages is left as an exercise for the reader.

References

[1] Kernighan B., Ritchie D., The C Programming Language, Prentice Hall (1988).
[2] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide; http://developer.intel.com/products/processor/manuals/index.htm
[3] Bovet D. P., Cesati M., Understanding the Linux Kernel, O'Reilly (2005).
[4] Gorman M., Understanding the Linux Virtual Memory Manager, Prentice Hall (2004).
[5] Aleph One, Smashing The Stack For Fun And Profit, Phrack #49; http://www.phrack.org/issues.html?issue=49&id=14
[6] GNU C Library, heap manager implementation source code; http://sourceware.org/git/?p=glibc.git;a=tree;f=malloc
[7] Solar Designer, JPEG COM Marker Processing Vulnerability; http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability
[8] Dowd M., McDonald J., Schuh J., The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities, Addison-Wesley Professional (2006).
[9] Nominations for Pwnie Awards (2008); http://pwnies.com/archive/2008/nominations/
[10] Project Valgrind web site; http://valgrind.org/
[11] Debian Security Advisory: DSA-1571-1, openssl – predictable random number generator; http://www.debian.org/security/2008/dsa-1571
[12] tmbinc blog post: Thank you, Datel; http://debugmo.de/2008/03/thank-you-datel/
[13] NIST.gov, Cryptographic hash algorithm competition; http://csrc.nist.gov/groups/ST/hash/sha-3/index.html
[14] Fortify Software Inc., NIST SHA-3 Competition Security Audit Results; http://blog.fortify.com/repo/Fortify-SHA-3-Report.pdf
[15] The Matrix Reloaded (2003); http://www.imdb.com/title/tt0234215/
[16] BindView advisory: sshd remote root (bug in deattack.c); http://www.mail-archive.com/bugtraq@securityfocus.com/msg04399.html
[17] Common Weakness Enumeration, CWE-190: Integer Overflow or Wraparound; http://cwe.mitre.org/data/definitions/190.html
[18] OpenSSH Security Advisory (adv.channelalloc); http://www.openbsd.org/advisories/ssh_channelalloc.txt
[19] Mozilla Foundation Security Advisory 2007-06, Mozilla Network Security Services (NSS) SSLv2 buffer overflows; http://www.mozilla.org/security/announce/2007/mfsa2007-06.html