Welcome to

7. Software Programming & Memory Corruption

KEA Kompetence OB2 Software Security

Henrik Kramselund he/him han/ham hlk@zencurity.com @kramse

Slides are available as PDF, kramse@Github

7-sw-programming-and-mem-corruption.tex in the repo security-courses

Plan for today

Subjects

- C language issues
- Memory Corruption Errors
- Buffer overflows, stack errors

Exercises

- Writing and exploiting a small buffer overflow
- Run debugger
- Pointers and Structure padding

Today, PLEASE work together, binary exploitation is not easy to grasp the first time! Expect to be frustrated

Reading Summary

Browse if you need to, many pages

If you have it *Gray Hat Hacking* chapters 1-2, 11-13 - browse if you need to, many pages.

Browse: *Smashing The Stack For Fun And Profit*, Aleph One, *Bypassing non-executable-stack during exploitation using return-to-libc* by c0ntex,*Basic Integer Overflows* by blexim.

Goals:

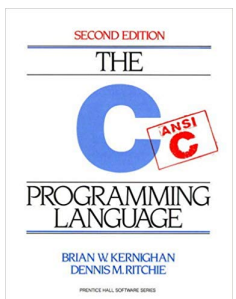
Understand more C, and problems associated with C

What is C

C (/si /, as in the letter c) is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, while a static type system prevents unintended operations. By design, C provides constructs that map efficiently to typical machine instructions and has found lasting use in applications previously coded in assembly language. Such applications include operating systems and various application software for computers, from supercomputers to embedded systems. **C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to make utilities running on Unix.** Later, it was applied to re-implementing the kernel of the Unix operating system.[6] During the 1980s, C gradually gained popularity. **Nowadays, it is one of the most widely used programming languages**,[7][8] with C compilers from various vendors available for the majority of existing computer architectures and operating systems. C has been standardized by the ANSI since 1989 (see ANSI C) and by the International Organization for Standardization.

Source: https://en.wikipedia.org/wiki/C_(programming_language)

C language issues



- C was very portable
- C was standardized
- Still lots of unspecified behaviour / undefined behaviour and implementation-defined
- So C on one operating system can be very different!
- Writing portable and correct code is hard!

https://en.wikipedia.org/wiki/Unspecified_behavior

Hello World

> While small test programs have existed since the development of programmable computers, the tradition of using the phrase "Hello, World!"as a test message was influenced by an example program in the seminal 1978 book The C Programming Language.[3] The example program in that book prints "hello, world", and was inherited from a 1974 Bell Laboratories internal memorandum by Brian Kernighan, Programming in C: A Tutorial:[4]

```
main( ) {
        printf("hello, world\n");
}
```

- Very compact, easily readable - more than assembler anyway!

Code and quote from https://en.wikipedia.org/wiki/%22Hello,_World!%22_program

Hello World

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

int main(int argc, char **argv)
{
        (void) argc;
        (void) argv;

        printf("Hello world!\n");

        return EXIT_SUCCESS;
}
```

- Dont forget to define function return value, your variables and arguments to the program and functions
- Dont forget to return something useful
- Whats an int anyway? integers

Source: a better hello world from https://da.wikipedia.org/wiki/Hello_world-program#C

Data types

- Character types char, signed char, unsigned char
- Integer types, short int, int, long int, long long int - plus unsigned
- Floating types float, double, long double
- Bit fields a specific number of bits in an object
- Signed and unsigned!
- Byte order, little or big endian - named from 1726 novel Gulliver's Travels by Jonathan Swift
- Big endian also called network order, as used in TCP/IP suite
- Lots more types we wont discuss in all details

Data types have a *range of values*

See more specific at https://en.wikipedia.org/wiki/C_data_types
also interesting is the float format
https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Example integer overflow

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{       (void) argc;  (void) argv;
                short int i1 = 32767;
                printf("First debug int is %d\n", i1);
                i1++;
                printf("Second debug int is now %d \n", i1);
}
```

```
user@Projects:programs$ gcc -o int1 int1.c && ./int1
First debug int is 32767
Second debug int is now -32768
```

We wont do the match in binary!

Compiling C code

```
  user@Projects:~$ mkdir openssh ; cd openssh
  user@Projects:openssh$ wget https://cdn.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-8.0p1.tar.gz
  --2019-09-16 20:52:13--  ...
  2019-09-16 20:52:14 (5.13 MB/s) - ‘openssh-8.0p1.tar.gz’ saved [1597697/1597697]
  user@Projects:openssh$ wget https://cdn.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-8.0p1.tar.gz.asc
  --2019-09-16 20:52:18--  ...
    2019-09-16 20:52:18 (26.5 MB/s) - ‘openssh-8.0p1.tar.gz.asc’ saved [683/683]
  user@Projects:openssh$ gpg --verify openssh-8.0p1.tar.gz.asc
  gpg: assuming signed data in 'openssh-8.0p1.tar.gz'
  gpg: Signature made Thu 18 Apr 2019 12:54:07 AM CEST
  gpg:                using RSA key 59C2118ED206D927E667EBE3D3E5F56B6D920D30
  gpg: Can't check signature: No public key
// fix by getting key etc.
```

user@Projects:openssh$ tar zxf openssh-8.0p1.tar.gz
user@Projects:openssh$ cd openssh-8.0p1
user@Projects:openssh-8.0p1$ ./configure && make && sudo make install

Summary compiling software

- Very often Linux/Unix software was distributed as archives - tgz - tar gzipped
- Configure checks for libraries and such things, see GNU Autoconf
- Tries to discover which options are required, which other libraries are available etc.
- Today we mostly use package managers with binary packages and signatures
- Preferred to use binary package systems like Debian apt!

Overiding types, type conversion

`Unsigned 00000001 is the same as 0000000000000001`

- Value-preserving - there is space in the new type to store all values
- Value-changing - new type cannot represent the same values, damn!
- To a wider type, we can *sign extend* for signed types and *zero extend* for unsigned types
- To a more narrow, need to truncate - bye bye bits, loose precision!
- Again, we wont repeat all the bit values and arithmetics, and how C extends the values
- Casting can be used to specify the explicit type conversion
  `(unsigned char) bob` - treat variable bob as an unsigned char type

Audit tip: The compiler can optimize out certain conversions! This can affect both cryptographic processes and other things! Beware

Type Conversion Vulnerabilities

```
int copy(char *dst, char *src, unsigned int len)
{
  while (len--)
    *dst++ = *src++
}
```

- Signed/unsigned conversions
- If you pass signed int to this, a conversion might result in a large value insted
- This large value will most likely result in overflow
- Most libc routines that take a size have `size_t` which is unsigned!
- Signed negative values become very large unsigned

Lets browse chapter 2 programming

Book examples show small programs, hello world style

- Introduces control structures, if/then/else, for loops, while loops
- Variables and arithmetics - including comparison
- Functions - in deep details with registers and stack pointers
- Talks about compilation and assembly - same process is done in interpreted languages before instructions run
- Strings are introduced, with types and pointers, hard concepts
- Format strings, how to print a string with printf
- The stack is described in detail, important for security/exploiting
- Above are the most important parts of the chapter, IMHO

Exercise

Now lets do the exercise

⚠ Small programs with data types 15min

which is number 22 in the exercise PDF.

Pointers

- Copy example already used pointers
- They point to a location in memory - a variable, a structure, some object of some kind
- Pointer arithmectic is dangerous
- ... but often used for reading structures, from the network
- Mismatched values, wrong asumptions, may result in pointers going wrong

Suricata VXLAN decode - defensive checks

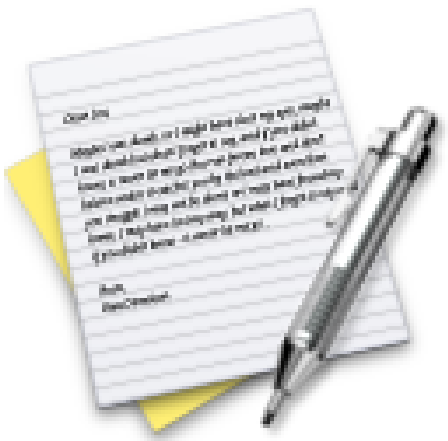
```
int DecodeVXLAN(ThreadVars *tv, DecodeThreadVars *dtv, Packet *p,
        const uint8_t *pkt, uint32_t len, PacketQueue *pq)
{
    if (unlikely(!g_vxlan_enabled))
        return TM_ECODE_FAILED;
    if (len < (sizeof(VXLANHeader) + sizeof(EthernetHdr)))
        return TM_ECODE_FAILED;
    const VXLANHeader *vxlanh = (const VXLANHeader *)pkt;
    if ((vxlanh->flags[0] & 0x08) == 0 || vxlanh->res != 0) {
        return TM_ECODE_FAILED;
    }
#if DEBUG
    uint32_t vni = (vxlanh->vni[0] << 16) + (vxlanh->vni[1] << 8) + (vxlanh->vni[2]);
    SCLogDebug("VXLAN vni %u", vni);
#endif
    StatsIncr(tv, dtv->counter_vxlan);
```

Source: excerpt from https://github.com/OISF/suricata/blob/master/src/decode-vxlan.c
Dont read past end of buffer!

Structures

- Pointers and Structure padding
- Lets look at C code from Suricata or Zeek, you choose
- Look how the structures for packets are defined

Exercise



Now lets do the exercise

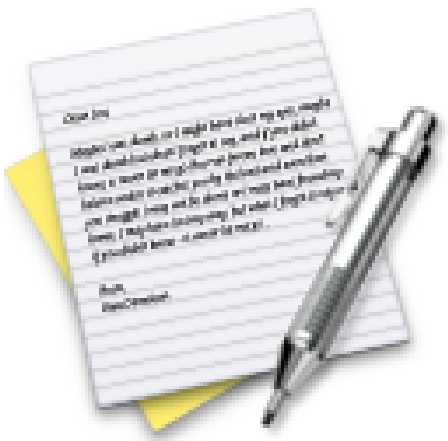
⚠ Pointers and Structure padding 30min

which is number **23** in the exercise PDF.

Look at the Wireshark 3.3 Packet Diagram

- Wireshark 3.3 has a packet diagram, which might be quite handy!
- Lets look into it, and if possible run a couple of network dumps
- Hopefully it will illustrate how network data is transferred
- Maybe install the newest version on your laptop

Exercise



Now lets do the exercise

Wireshark 15 min

which is number **24** in the exercise PDF.

Lets browse chapter 3 Exploiting

Book examples show

- Buffer overflow basics, bad comparison in first pages, see real life example later in this slideshow
- Shellcode included as a string-buffer, page 121
- NOP sled in the same example
- Stack based buffer overflow
- Using Perl for generating long sequences, `print "A"x20;`
- Environment variables and using them for storing data aka shellcode
- Heap based buffer overflow
- Overwriting function pointers
- Format string overflows
- Above are the most important parts of the chapter, IMHO

You dont need to remember it all!

Real vulns

> Hold tight, this may blow your mind... A low-privileged user account on most Linux operating systems with UID value anything greater than 2147483647 can execute any systemctl command unauthorizedly—thanks to a newly discovered vulnerability. The reported vulnerability actually resides in PolicyKit (also known as polkit)—an application-level toolkit for Unix-like operating systems that defines policies, handles system-wide privileges and provides a way for non-privileged processes to communicate with privileged ones, such as "sudo,"that does not grant root permission to an entire process. The issue, tracked as **CVE-2018-19788**, impacts PolicyKit version 0.115 which comes pre-installed on most popular Linux distributions, including Red Hat, Debian, Ubuntu, and CentOS. ... Where, `INT_MAX` is a constant in computer programming that defines what maximum value an integer variable can store, which equals to 2147483647 (in hexadecimal 0x7FFFFFFF).

- Reality bites again:
  https://thehackernews.com/2018/12/linux-user-privilege-policykit.html
- *Red Hat has recommended system administrators not to allow any negative UIDs or UIDs greater than 2147483646 in order to mitigate the issue until the patch is released.*

Memory Corruption Errors

- Assume all memory corruption vulnerabilities should be treated as exploitable, until you can prove otherwise
- Auditting and exploit creation are different, but highly complementary skills

- Buffer overflows, stack errors
- We have already worked through our buffer overflow example, but chapter 3 describes stacks in more detail
- Chapter also describes how functions are called and data is stored in *stack frames*
- Of interest is the off-by-one errors on page 180, how a single byte overwrite can result in exploitation
- Heap overflows

Shell code

```
char *args[] = { "/bin/sh", NULL };

execve("/bin/sh", args, NULL);
```

- The concept of shell code is explained in detail
- Shell code can be quite small, smallest seems to be 21 bytes for Linux x86

Integer overflows

```
----[ 1.2 What is an integer overflow?
Since an integer is a fixed size (32 bits for the purposes of this paper),
there is a fixed maximum value it can store.  When an attempt is made to
store a value greater than this maximum value it is known as an integer
overflow.  The ISO C99 standard says that an integer overflow causes
"undefined behaviour", meaning that compilers conforming to the standard
may do anything they like from completely ignoring the overflow to aborting
the program.  Most compilers seem to ignore the overflow, resulting in an
unexpected or erroneous result being stored.

----[ 1.3 Why can they be dangerous?
Integer overflows cannot be detected after they have happened, so there is
not way for an application to tell if a result it has calculated previously
is in fact correct.  This can get dangerous if the calculation has to do
with the size of a buffer or how far into an array to index.  Of course
most integer overflows are not exploitable because memory is not being
directly overwritten, but sometimes they can lead to other classes of bugs,
frequently buffer overflows.  As well as this, integer overflows can be
difficult to spot, so even well audited code can spring surprises.
```

Source: *Basic Integer Overflows* by blexim http://www.phrack.com/issues.html?issue=60&id=10#article

Integer overflows

```
----[ 1.2 What is an integer overflow?
Since an integer is a fixed size (32 bits for the purposes of this paper),
there is a fixed maximum value it can store.  When an attempt is made to
store a value greater than this maximum value it is known as an integer
overflow.  The ISO C99 standard says that an integer overflow causes
"undefined behaviour", meaning that compilers conforming to the standard
may do anything they like from completely ignoring the overflow to aborting
the program.  Most compilers seem to ignore the overflow, resulting in an
unexpected or erroneous result being stored.

----[ 1.3 Why can they be dangerous?
Integer overflows cannot be detected after they have happened, so there is
not way for an application to tell if a result it has calculated previously
is in fact correct.  This can get dangerous if the calculation has to do
with the size of a buffer or how far into an array to index.  Of course
most integer overflows are not exploitable because memory is not being
directly overwritten, but sometimes they can lead to other classes of bugs,
frequently buffer overflows.  As well as this, integer overflows can be
difficult to spot, so even well audited code can spring surprises.
```

Source: *Basic Integer Overflows* by blexim http://www.phrack.com/issues.html?issue=60&id=10#article

Return-to-libc

```
How does the technique look on the stack - a basic view will be something
similar to this:
[-] Buffer overflow smashing EIP and jumping forward to shellcode
                                                1                    2
|------------------|-----------|------------|--------------------------|
|             AAAAAAAAAAAA      |    RET     |        SHELLCODE         |
|------------------|-----------|------------|--------------------------|
                   args              EBP        EIP
[-] Buffer overflow doing return-to-libc and executing system function
                                                  1             2         3
|------------------------------|------------|--------------|------------|
|            buffer            |   system   |   fake_ret   |   /bin/sh  |
|------------------------------|------------|--------------|------------|
                   args              EBP        EIP
```

Instead of putting code on the stack, that cannot be executed, put on a fake return address which goes to a function in the C library, like `system("/bin/sh")`

Source: *Bypassing non-executable-stack during exploitation using return-to-libc* by c0ntex | c0ntex[at]gmail.com

Return-oriented programming (ROP)

Nogle ting bliver også sværere - buffer overflow protection

Teknologier som Address Space Layout Randomization ASLR
http://en.wikipedia.org/wiki/Address_space_layout_randomization

No eXecute NX-bit, dele af memory kan ikke afvikles som kode

Data Execution Prevention DEP
http://en.wikipedia.org/wiki/Data_Execution_Prevention

Modsvar: Return-oriented programming (ROP) is one of the buzzing advanced exploitation techniques these days to bypass NX, ASLR - byg exploits med stumper af eksisterende kode og stakken

Kilder: diverse præsentationer fra BlackHat
http://www.blackhat.com/html/bh-us-10/bh-us-10-archives.html
https://media.blackhat.com/bh-us-10/presentations/Zovi/BlackHat-USA-2010-DaiZovi-Return-Oriented-Exploitation-slides.pdf

Return Oriented Programming

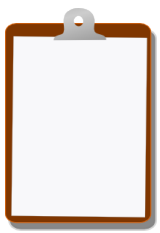
- By doing *return chaining* build shell-code from existin program
- Instead of returning to functions, return to instruction sequences followed by a return instruction
- Can return into middle of existing instructions to simulate different instructions
- All we need are useable byte sequences anywhere in executable memory pages
- Scan executable memory regions of common shared libraries for useful instructions followed by return instructions
- Chain returns to identified sequences to form all of the desired gadgets from a Turing desired gadgets from a Turing-complete gadget catalog complete gadget catalog. The gadgets can be used as a backend to a C compiler

https://media.blackhat.com/bh-us-10/presentations/Zovi/BlackHat-USA-2010-DaiZovi-Return-Oriented-Exploitation-slides.pdf

Function call return

- Function calls are implemented using stacks, and they usually return
- By finding functions that do something small, and then return an exploit might be able to pack a stack, and return through an exploitation technique known as *Return Oriented Programming* (ROP)
- Note: there is a paper on the schedule Removing ROP Gadgets from OpenBSD Todd Mortimer mortimer@openbsd.org, can be downloaded at https://www.openbsd.org/papers/asiabsdcon2019-rop-paper.pdf
- This changes the rules, by changing the entry (prologue) and return from functions (epilogue): RETGUARD is a mechanism that adds instrumentation to the prologue and epilogue of each function that terminates in a return instruction.

For Next Time



Think about the subjects from this time, write down questions

Check the plan for chapters to read in the books

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools