Simple Code Injection Techniques for ELF

COMP3703 Software Security

Slides prepared by H. Gunadi and A. Tiu, based on Chapter 7 of Andriesse's "Practical Binary Analysis", No Starch Press, 2019.

Outline

- Bare-metal binary modification using hex editing
- Modifying shared library behaviour using LD_PRELOAD
- Injecting a code section
- Calling the injected code by hijacking:
  - Entry point modification
  - Constructor / destructor
  - GOT
  - PLT
  - A function call in the .text section
- Live demonstration of code injection techniques

Binary modification with hex editing

- If we know wher we can directly edit it using a hex editor, e.g., n Linux.
  - Use the help of disassembler to find the correct location.
- It's simple and requires only basic tools.
- However, it only allows in-place editing (not adding new bytes).
  - It's difficult (or even impossible) to correctly identify and fix all the broken references.
  - We can rewrite padding bytes, dead code, or unused data.

Binary modification with hex editing

- Still, there are st[illegible] uses, e.g.,:
  - disabling malwa[illegible] ebugging techniques by rewriting the check with NOPs.
  - Fix simple bugs, e.g., off-by-one error.
  - It might even be practical in the real world! (https://blog.0patch.com/2021/03/remotely-exploitable-0day-in-internet.html)
- References for the opcodes and operand format for x86 instructions:
  - http://ref.x86asm.net.
  - http://ref.x86asm.net/coder64-abc.html (for x86-64)

Example: off-by-one bug

- Easy to miss in s[illegible]opying.
  - We often forget [illegible]minator.
  - E.g., when we use a safer version of strcpy (strncpy)
  - See: https://cwe.mitre.org/data/definitions/193.html
- In our example, we have a simple program that encrypt a file using simple xor operations.
- But it contains an off-by-one bug; the last byte is not processed.

Example: off-by-one bug

- Assume we do not have the source code.
- Use disassembler to find where the loop condition is enforced.
  - Then change the comparison instruction used for the loop condition.
- Alternatively, modify the value of the guard.

```
n = .. // length of buffer
...
j = 0;
for(i = 0; i < n-1; i++) {
  buf[i] ^= key[j];
  j = (j+1) % strlen(key);
}
...
```

Example: off-by-one bug

```
n = .. // length of buffer
...
j = 0;
for(i = 0; i < n-1; i++) {
  buf[i] ^= key[j];
  j = (j+1) % strlen(key);
}
...
```

Original binary:

```
401188:       41 0f b6 04          movzx  eax,BYTE PTR [r15+rdx*1]
40118d:       30 03                xor    BYTE PTR [rbx],al
40118f:       4c 89 ff             mov    rdi,r15
401192:       48 8d 6a 01          lea    rbp,[rdx+0x1]
401196:       48 83 c3 01          add    rbx,0x1
40119a:       e8 c1 fe ff ff       call   401060 <strlen@plt>
40119f:       31 d2                xor    edx,edx
4011a1:       49 89 c0             mov    r8,rax
4011a4:       48 89 e8             mov    rax,rbp
4011a7:       49 f7 f0             div    r8
4011aa:       49 39 dd             cmp    r13,rbx
4011ad:       75 d9                jne    401188 <main+0xb8>
```

- jne <dest>: Jump to rip+<dest> if the ZF flag is not set (so (r13-rbx) != 0).
- Opcode for jne: 75.

Example: off-by-one bug

```
n = .. // length of buffer
...
j = 0;
for(i = 0; i <= n-1; i++) {
  buf[i] ^= key[j];
  j = (j+1) % strlen(key);
}
...
```

Modified binary:

```
401188:       41 0f b6 04       movzx  eax,BYTE PTR [r15+rdx*1]
40118d:       30 03             xor    BYTE PTR [rbx],al
40118f:       4c 89 ff          mov    rdi,r15
401192:       48 8d 6a 01       lea    rbp,[rdx+0x1]
401196:       48 83 c3 01       add    rbx,0x1
40119a:       e8 c1 fe ff ff    call   401060 <strlen@plt>
40119f:       31 d2             xor    edx,edx
4011a1:       49 89 c0          mov    r8,rax
4011a4:       48 89 e8          mov    rax,rbp
4011a7:       49 f7 f0          div    r8
4011aa:       49 39 dd          cmp    r13,rbx
4011ad:       73 d9             jae    401188 <main+0xb8>
```


- Change `jne <dest>` to `jae <dest>`: Jump if "above" or "equal" -- jump to <dest> if (r13 >= rbx).
- Opcode for `jae`: 73.

Binary Modification Using LD_PRELOAD

- Hex editing is tedious, error prone, and quite restrictive.
- We can use LD_PRELOAD to modify the behaviour of shared library functions.
- LD_PRELOAD is an environment variable to specify one or more libraries for the linker to load ***before*** any other libraries.
- Functions in the pre-loaded libraries take precedence over functions of the same name in libraries loaded later.
  - E.g., if a preloaded library has a function called printf, then that version of printf is the one used at runtime, not libc's printf.

Binary Modification Using LD_PRELOAD

To override a share function, say libc's malloc, used in a target bir ;

- Create a custom library, and define a function called malloc (with the same function signature as the original malloc) in that library.
- Set the LD_PRELOAD to point to the path of the custom library before running the target binary.
  - The custom malloc will be called instead of the libc's version.
- Optionally, the custom malloc may also invoke the original malloc.
  - This needs a mechanism to refer to the original malloc (using the *dlsym* function) – see "Practical Binary Analysis" (Chapter 7) for details.

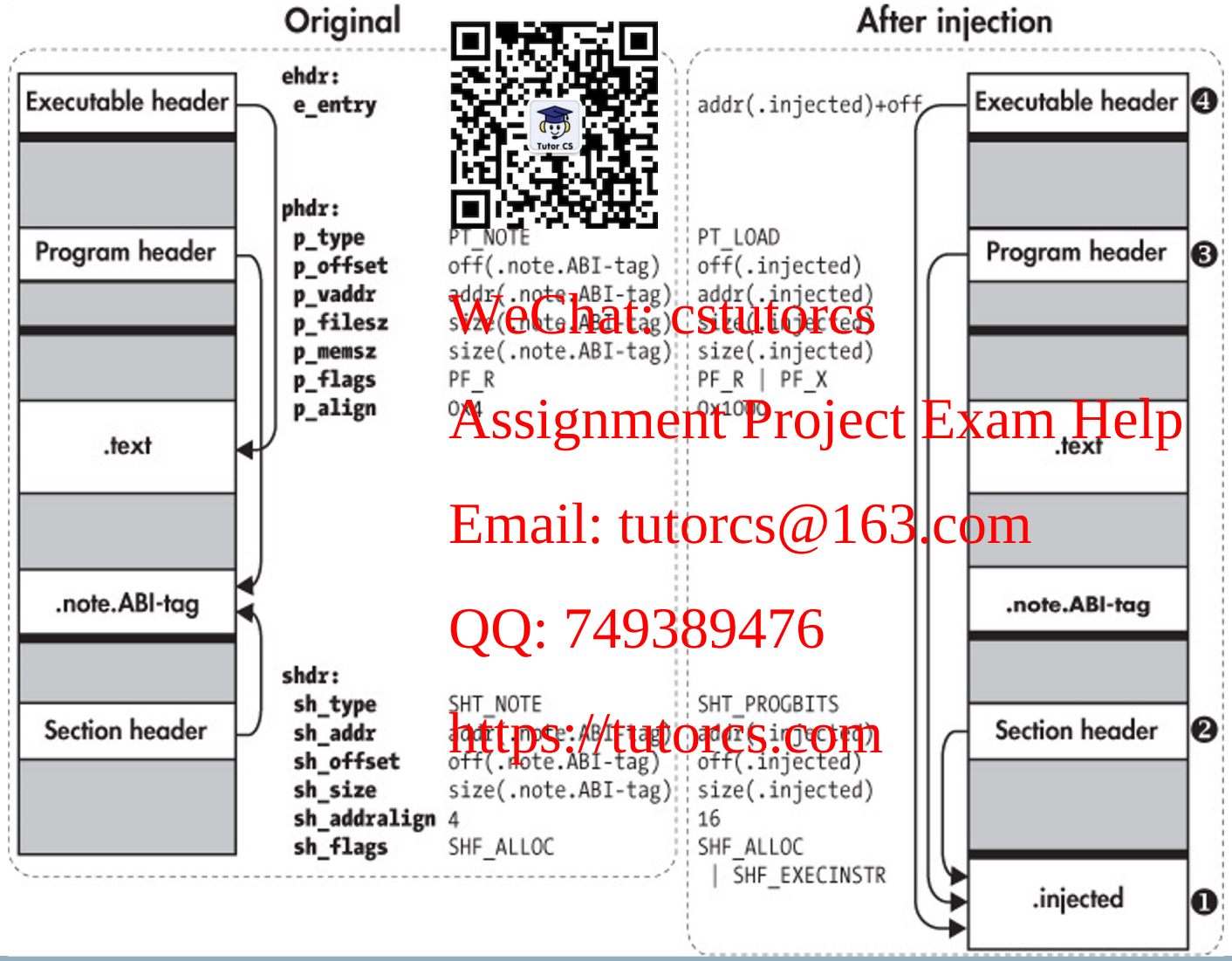
Injecting a code section

- The LD_PRELOAD is limited to modify behaviours of shared library functions.
- A more powerful technique is to inject arbitrary code into a target ELF binary.
- This is significantly more complicated – we need to make sure not to break various dependencies in the binary.
- We will be using a tool called elfinject.
  - Appendix B of "Practical Binary Analysis" explains elfinject in detail.

Injecting a code section: overview

- Inject the code in a new code section (rather than modifying an existing section),
  - This avoids having to update targets of jumps in the existing code.
  - The new section can be added at the end of the file: a section in ELF can be located anywhere in the file, as long as its section header contains the correct offset and size.
- Create (reuse) a section header.
- Create (reuse) a program header.
- Add/rename the section name in .shstrtab.
- Call the injected code.

Injecting a code section: overview



Program headers: add or reuse?

- Programs headers are located in between the executive headers and the sections.
- Adding a new program header potentially shifts the offsets of the sections (and section headers), requiring a remapping of virtual addresses and file offsets.
- Avoid adding program headers if possible; instead, reuse existing one.

Reusing program header

- The PT_NOTE header is always safe to overwrite.
  - PT_NOTE segments contains optional information, e.g., GNU/Linux binary, kernel version expected, etc.
- Required modifications (done by elfinject):
  - Change its type to PROGBITS, i.e., containing code.
  - Change the sh_addr, sh_offset, and sh_size fields to describe the new .injected section.
  - Change sh_addralign to 16 bytes to ensure that the code will be properly aligned when loaded into memory.
  - Mark the segment as executable.

Calling the injected code

Once we put the injected code, there are several alternatives to call it:

- Entry Point Modification
- Hijacking Constructors and Destructors
- Hijacking GOT Entries
- Hijacking PLT Entries
- Redirecting Direct and Indirect Calls

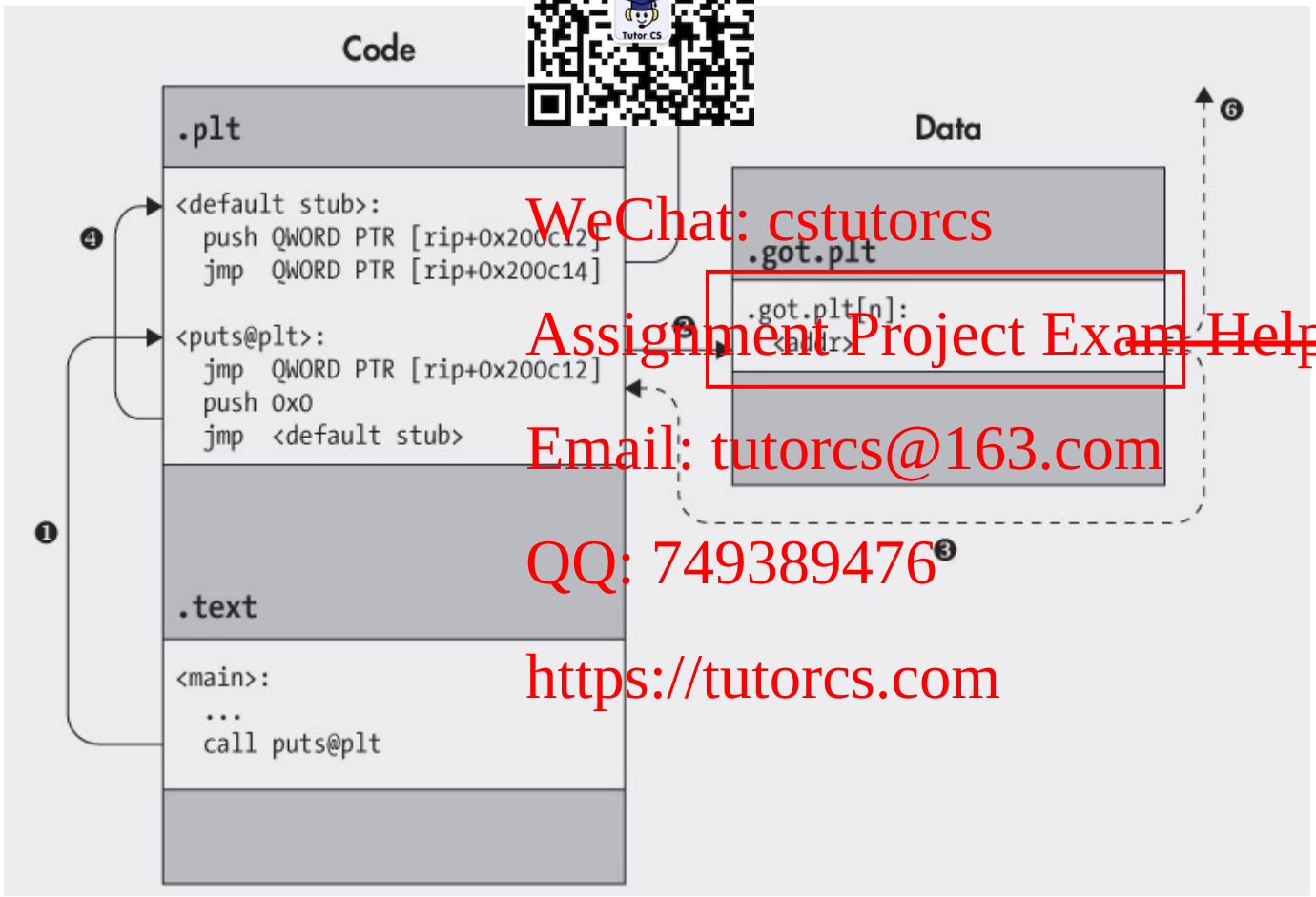
Calling the injected code: Entry point modification

- Modify the entry point to jump to the injected code.
  - Modify the e_entry value of the ELF header.
  - The injected code will run when the program starts.
- We can then jump to the original entry point to resume execution.

Hijacking GOT entries

- Replace the GO[illegible]n .got.plt section in the ELF binary) for a libra[illegible]on we want to modify with the address of the injected code.
  - The injected code will be called every time the library function is called.
- .got.plt is writable during program execution, so can also be dynamically hijacked.

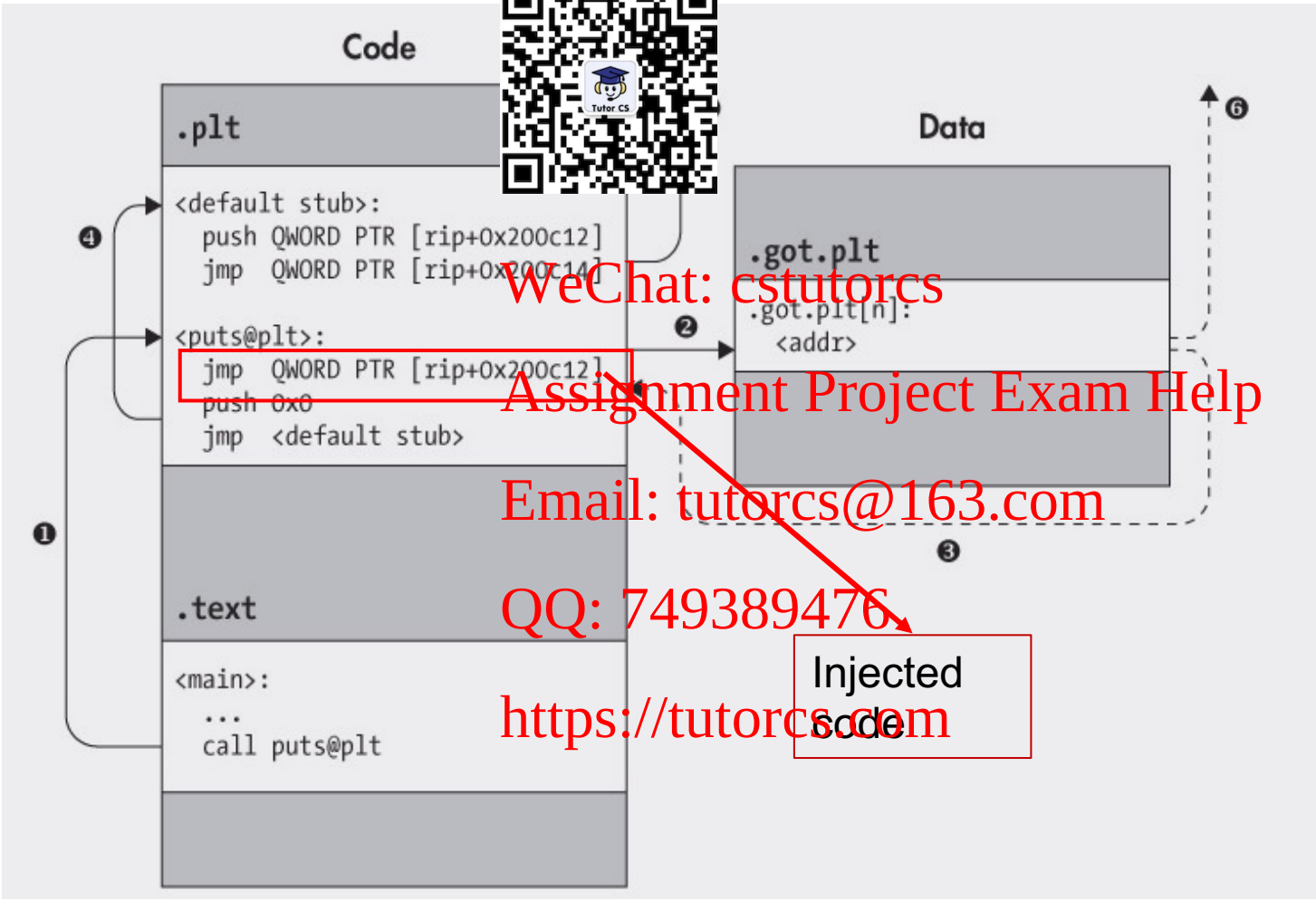
Hijacking GOT entries



Hijacking PLT entries

- Similar to hijacki entries, we modify the PLT stub to point to our in nction.
- What we modify is the target of the jump instruction.
- This may mess up the rest of the instructions in the modified PLT stub
  - But it does not matter because they will not be executed.

Hijacking PLT entries



Redirecting direct and indirect calls

- Hijacking GOT a[illegible] only work for library calls.
- We can directly modify the direct call instructions in the main program body.
- In the case of indirect call, there are some issues:
  - Need to find the address of the indirectly called function.
  - Encoding of the direct call may be longer, so replacing the indirect call with a direct call may shift the offset of other instructions.

Section headers: add or reuse?

- Since section headers are located at the end of the ELF binary, adding a new header is straightforward – no need to adjust other parts of the file.
- Reusing a section header may be preferrable.
  - One candidate is the header for section .note.ABI-tag.

Some notes on PIE

- Recall that in shared libraries, the address of a function can only be resolved at runtime (e.g., via lazy binding using PLT/GOT).
- This can be applied to an executable, resulting in Position Independent Executables (PIE).
  - The entry point (and address of all functions) in PIE are resolved at runtime.
- Some techniques that use injected code with direct jumps will no longer work, since the target address maybe relocated at runtime.

Summary

- We have seen s[illegible]ple techniques for code injection.
- These techniques do not modify headers/sections length – thus adjustments to existing instructions are localised.
- More advanced techniques will be covered in later parts of this course.