Binary Reverse Engineering

with Ghidra

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
X/GitHub/. . . : zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

Outline

Reversing

How can we know what a program *really* does?
In particular, a binary program, without anything else
No source code, no documentation, no symbols, ...

Reverse engineering (AKA *reversing*) consists in attempting to understand how a device/process/piece-of-software accomplishes a task

- in our case, what a program does, and how
- can we just use ChatGPT/CoPilot/*some Large Language Model*...?

Reversing challenges

- low-level languages
- often, no “symbols”; i.e. meaningful names
- (almost) no type information
- no class/module/namespace boundaries
- code and data can be mixed, and they usually do
- difficulty in adding/changing/removing instructions
- . . .

With some effort and the right tools, we’ll break programs apart to learn how they work and what they do.

An old saying goes:
Once you speak machine code, then every program becomes open-source ☺

Beware

We always reverse engineer programs that have been explicitly written to be reversed (the so-called "crackmes", and some CTF-like challenges), open-source software or malware

- It is not easy to find detailed and practical advice about the legal boundaries of reversing copyrighted software in EU, US, ... worldwide
- As far as I understand (but I am not a lawyer), in EU you are allowed to reverse engineer a program for private purposes or to understand its interfaces to allow another program to interface with it

Never run unknown files

Unless you're 100% confident they are safe

Exercises/assignments, not clearly marked as malware, are safe (AFAIK ☺)

Static vs Dynamic Analysis

Broadly speaking, we can split our methods/tools into

- Dynamic analysis: we run the binary and analyze it, or log its behavior, as it executes
  - often simpler, can observe runtime states
  - can be harmful; e.g., malware
  - not everything is necessarily apparent
  - for each run you observe *that* particular execution, and might miss *interesting* parts of the code; e.g.

```
if (random()==0xcafebabe) { /* interesting stuff */ }
```

- Static analysis: we reason about the binary without running it
  - you can analyze the whole binary in one go
  - you don't need a CPU/system that can run such a binary
  - (obviously, almost) no knowledge of runtime states
  - can be difficult to pinpoint *interesting* parts

First approach: identification/integrity checking

How can we check the "identity" of a file?

Hash functions:

controlli l'hash, ma possono essere manipolati, utilizzando certi metodi
diversi file possono avere lo stesso hash

Linux/WSL	`md5sum`, `sha*sum`, ...
Windows	HashMyFiles `https://www.nirsoft.net/utils/hash_my_files.html`
	...

→`sample{1,2}.elf`

Format and strings

The first approach usually consists in checking

Formats

- `file`
- `magika` `https://github.com/google/magika`
- `diec` `https://github.com/horsicq/Detect-It-Easy`
- `polyfile` `https://github.com/trailofbits/polyfile`
- ...
- when in doubt, hex editors (e.g., ImHex `https://github.com/WerWolv/ImHex`)

Strings

- `strings`; in Windows, *SysInternals Suite* `https://docs.microsoft.com/en-us/sysinternals` or inside the MS Store
- `floss` `https://github.com/mandiant/flare-floss`
- ...

Executable specific tools

ELF `hte`, `readelf`, `objdump`, `nm`, ...

- XELFViewer
  `https://github.com/horsicq/XELFViewer`

PE `hte`, `dumpbin.exe`, ...

- PE Bear
  `https://github.com/hasherezade/pe-bear`
- PE Studio
  `https://www.winitor.com/download`
- XPEViewer
  `https://github.com/horsicq/XPEViewer`
- ...

Linux: beware of 32-bit executables in modern distros

Modern Linux distros do not ship 32-bit libraries by default; e.g., on 64-bit Ubuntu systems, you need to add those:

- `sudo dpkg --add-architecture i386`
- `sudo apt update`
- `sudo apt install libc6-dbg:i386`

A (simpler) alternative seems to be installing the package `gcc-multilib`

Other distributions should provide analogous packages.

Demo/exercise: restricted-area v1.0

CTFs and Flags

All challenge descriptions are *tongue in cheek*: don't take them too seriously ☺ (but read them carefully, since they may contain hints)

In these exercises, *flags* are strings in the format `BASC{...}`.

> *A flag is trapped inside a restricted area, protected by a super-secure password. Will you find it?*

→ `restricted_area_v1`

Demo/exercise: restricted-area v2.0

We fixed all vulnerabilities of the previous version.
This program doesn't accept any password, so you can't guess them! Ahahah
Can you still manage to get the flag?

→ `restricted_area_v2`

Can you find the flag. . . with the tools discussed so far?

To go deeper, we need to look at the code. . .

How could we analyse/reverse machine code?

1. Disassemble, i.e. decoding bytes into assembler instructions
2. Group instructions into (basic) blocks and functions
3. Abstract into graphs
   - control-flow graphs describe the structure of functions
   - call graphs describe the relationships between functions
   - code/data cross references help in understanding dependencies and find interesting (starting) points, to explore further
4. Decompile, i.e. *trying* to recover corresponding C code

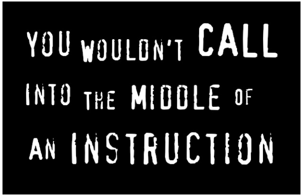
The general goal is inferring higher-level semantics, by giving meaningful names, and adding comments

- e.g. "increment variable `counter`" instead of `ADD [RSP-0x1C],1`
  or "call function `ask_password`" instead of `CALL 0x61a4be0032`

Disassembling

Disassembling might sound boring and trivial, however... [ACvdV+16, JZL+20]

- How do you distinguish between *code* and *data*?
  - E.g., compilers may embed data inside code sections
- Instructions could be *overlapped* [JLH13, LD03]



`https://twitter.com/awesomekling/status/1369178264716120065`

- Code can be *encrypted*/*packed*/*obfuscated*/...

Types of disassemblers

Static disassemblers can be split into

- Linear sweep
  - start at the beginning
    - disassemble the first instruction
    - then the following one,
    - and so on
  - no attempt to understand the control flow
- Recursive traversal
  - focus on of control flow
  - start at the entry point
    - if non-branch, continue to the next instruction
    - if branch, continue to possible targets

→ `asm-examples/tricky_disasm/tricky.asm`

Basic Blocks

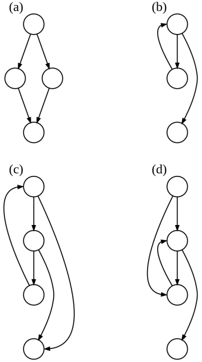
Basic blocks are sequences of instructions in which

- flow of control enters at the beginning
- leaves at the end, without branching (except at the end)

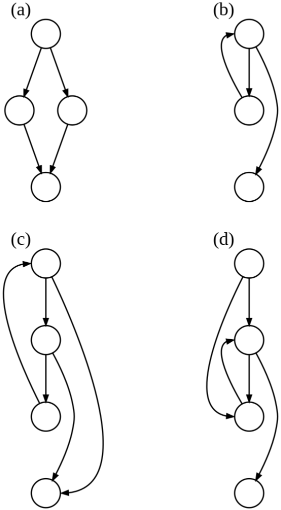
Control flow graph

A Control Flow Graph captures the execution paths that can take place inside a function

- the nodes are basic blocks
- edges represent potential control flow paths
    - back edges represent loops



CFGs



- (a) an if-then-else
- (b) a while loop
- (c) a loop with two exits, e.g. while with an if. . . break in the middle
- (d) a loop with two entry points, e.g. goto into a while or for loop

https://en.wikipedia.org/wiki/Control_flow_graph

CFG Construction challenges

Problematic constructs are:

- indirect jump instructions; mainly used to implement switch statements. [MM16] handles various kinds of jump tables with a backward intraprocedural dataflow analysis; e.g.:

Instruction	Jump target analysis
`mov %ebp,0xf8(%rsp)`	0 ≤%rdx==0xf8(%rsp)==%ebp≤ 5
`cmp $0x5,%ebp` `ja 43a4ab`	0 ≤%ebp≤ 5
`lea 0x525e8f(%rip),%rax` `lea -0x302(%rip),%rcx`	%rax=0x9602a0, %rcx=0x43a116 JTT=0x43a116+[0x9602a0+%rdx×8]
`movslq 0xf8(%rsp),%rdx`	%rdx==0xf8(%rsp)
`add (%rax,%rdx,8),%rcx`	JTT=%rcx+[%rax+%rdx×8]
`jmpq *%rcx`	JTT=%rcx

- non-returning functions; e.g. `exit` or `abort`. If not recognized, an nonexistent control flow is assumed from a non-returning call to its next block

CFG Partitioning challenges

Functions can share BBs, their code can be non-contiguous

```
35110db510 <__write>:
35110db510    cmpl $0x0,0x2b8199(%rip)
35110db517    jne 35110db529
35110db519 <__write_nocancel>:
35110db519    mov $0x1,%eax
...
35110db526    jae 35110db559
35110db528    retq
35110db529    sub $0x8,%rsp
...
35110db556    jae 35110db559
35110db558    retq
35110db559    mov 0x2b2a48(%rip),%rcx
...
35110db56c    jmp 35110db558
```

Figure 9: Functions sharing code and non-contiguous functions example from libc. The code in blue is shared by both functions. `__write_nocancel` is also a non-contiguous function, which is separated by the code from `__write`.

From: [MM16]

Tail calls: 2 or 3 functions?

```
BZFILE* BZ_API (BZ2_bzdopen) (int fd, char * mode)
 { return bzopen_or_bzdopen(NULL,fd,mode,1);}
BZFILE* BZ_API (BZ2_bzopen) (char *path, char * mode)
 { return bzopen_or_bzdopen(path,-1,mode,0);}
---
// entry point of bzopen_or_bzdopen, but no function symbol
351f40baa0   mov %rbx,-0x30(%rsp)
...
351f40bd70 <BZ2_bzdopen>:
351f40bd70   mov %rsi,%rdx          // set mode
351f40bd73   mov $0x1,%ecx          // set open_mode
351f40bd78   mov %edi,%esi          // set fd
351f40bd7a   xor %edi,%edi          // set path
351f40bd7c   jmpq 351f40baa0
...
351f40bd90 <BZ2_bzopen>:
351f40bd90   mov %rsi,%rdx          // set mode
351f40bd93   xor %ecx,%ecx          // set open_mode
351f40bd95   mov $0xffffffff,%esi   // set fd
351f40bd9a   jmpq 351f40baa0
```

Figure 10: A tail call example from bzip2. BZ2_bzdopen **and** BZ2_bzopen **both perform a tail call to the internal function** bzopen_or_bzdopen**, which does not have a function symbol.**

From: [MM16]

Finding function entry points (without symbols)

Without complete symbols, identifying function entry points becomes significantly more difficult; you can tackle this issue by

- recognizing *function prologue patterns*; *however*, difficult to
  - adapt to variations in compilers and optimization levels
  - detect non-standard/hand-written functions
- parsing `.eh_frame` section
  `https://bitlackeys.org/#eh_frame`
- using the CFG; see e.g. [ASB17]

(non-inlined) library functions are somewhat easier to detect:

- in dynamically linked programs, function names cannot be omitted
  - tricky programs may play with `dlsym`/`GetProcAddress` and/or *offsets*
- in statically linked programs we can try *fingerprinting* known functions [JRM11]

Call graphs

Once you have identified functions, then it can be useful representing relationships between functions in a Call Graph

- each node represents a function
- each edge (f, g) indicates that f calls g
  - A cycle indicates recursive calls
- can be computed statically or dynamically
  - E.g. a profiler could produce a dynamic call graph
- a static call graph represents every possible run
  - computing the exact static call graph is an undecidable problem
  - static call graph algorithms are generally overapproximations
    - when function pointers or virtual methods are invoked, a combination of *dataflow* and *data type* analysis can limit the set of potential *targets*
    - if the program loads code modules dynamically at runtime, there is no way to be sure that the control flow graph is complete

Outline

Getting started

- A highly extensible application for software reverse engineering
- Official site: `https://ghidra-sre.org/`
  You find there binaries and installation guide
- Guides/tutorials
  - `docs/GhidraClass`, inside Ghidra installation directory
  - these slides ☺
  - (paid, excellent) The Ghidra Book
    `https://nostarch.com/GhidraBook`
  - (free) "Hands-on Reversing with Ghidra"
    by Jeremy Blackthorne @ Ringzer0 Training 2022
    `https://vimeo.com/728095141`

  script snippets: `https://github.com/HackOvert/GhidraSnippets`
- The main, closed-source, alternatives are:
  - *IDA Pro* `https://hex-rays.com/ida-pro/`
  - *Binary Ninja* `https://binary.ninja/`

The environment

Let's use Ghidra to analyze `restricted_area_v1` ...

- Projects
  - Loaders
  - Tools → Code Browser
- Auto-analysis (with/w.o. the PDB file)
- Window handling/layout

Shortcuts and suggested options

Shortcut to remember:

- `F1`
- `F4` while the mouse is over any toolbar icon or menu item

Suggested key bindings (*Edit* → *Tool Options...* →*Key Bindings*):

- ESC/shift-ESC for *Previous/Next location in History*
- X/ctrl-X for *Find References To* and *Show References To Address*

Moreover, you may consider to set "Left" for *Mouse Button Activate* in *Listing Fields*→ *Cursor Text Highlight*

Symbol views

- *Symbol table* displays a tabular view of each symbol
  - *Symbol reference* display reference information for selected symbol, and type of reference

RW	read/write data access
Read	read-only data access
Write	write-only data access
Data	general data access
Branch	conditional jump
Jump	unconditional jump
Call	subroutine/function call
Unknown	all other reference types

- *Symbol tree* hierarchical view

Imported functions

These views are useful to quickly look at imports/exports

Default labels

Default label prefixes

- Common

  FUN a function
  DAT a data item (AKA "global variable")
  LAB code (usually, some jump-target inside a function)

- Others

  SUB code that has at least one call to it (but it doesn't seem a proper function)
  EXT an external entry point
  OFF the associated address is *offcut*, i.e. inside of an instruction or data item
  UNK none of the above

You can change/assign a label with `L`

Navigation

Views are synchronized

- G to jump to address/label (wildcards)/expression
- toolbar: next/previous
  - location
  - selected/highlighted range
  - code-unit
- double-click on label/address in the code browser
- click on names in *Functions*/...
  (when enabled, otherwise double-click)

"next"/"previous" meaning for I/D/U

When searching for Instructions/Data/Undefined items, Ghidra will skip all contiguous items of the same type. Then, it will search for the item.

Cross-references

- inside *Listing*

c	call
j	jump
*	pointer
w	write
r	read

- in *Reference to ...* views

Function calls

To display incoming and outgoing calls:

- *Function Call Graph*
- *Function Call Trees*

Searching

Search...

Program Memory performs searching for byte patterns in program memory

Program Text searches for text strings in various parts of the listing

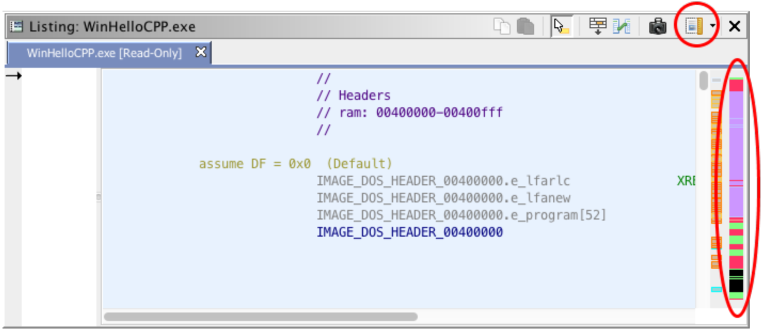
- supports wildcards but not full regular-expressions

For Strings finds potential strings within the program memory

For ... find scalars, instruction patterns, ...

Bars

- Navigation marker area (right-click to see/select categories)
- Overview bars
  - Entropy
  - Overview



Columns and Column-filters

- Some windows let you choose the columns and their filters (e.g. *Functions* and *Defined Strings*)
- Filters can be saved and reused later

Other views

- *Bytes* (click on its settings for adding "columns")
- *Defined data*
- *Defined strings*

View synchronization

Views are synchronized, you can make snapshots/independent-views with the "camera"

Startup code

When we analyzed `restricted_area_v1`, Ghidra found and jumped to `main`. Let's check `v2` out...

Most executable share the same startup code, which

1. initializes the CRT
   - in Windows the very first thing is initializing the security cookie
2. calls the main function (`main` or `WinMain`)
   - saving the return value *v*
3. cleans up the CRT
4. exits with exit-value *v*

The executable entry-point is called `entry`

`entry`

If Ghidra does not name `entry` automatically, you can find the entry-point by navigating the ELF/PE-header (see *Program trees* → *Headers*)

Linux/Windows: `main`

According to the C standard,

```
int main(void);
int main(int argc, char *argv[]);
```

however, a common extension is

```
int main(int argc, char *argv[], char *envp[]);
```

Decompiler Parameter ID analysis

While generally useful (but expensive: turned-off by default for large programs), this analysis can be misleading when looking for `main`/`WinMain` because it can "hide" parameters, at call site, that are unused by the function

Windows: WinMain

https://learn.microsoft.com/en-us/windows/win32/learnwin32/winmain--the-application-entry-point:

```
int __stdcall WinMain(
  [in] HINSTANCE hInstance,
  [in] HINSTANCE hPrevInstance,
  [in] LPSTR     lpCmdLine,
  [in] int       nShowCmd
);
```

- hInstance is actually the module load-address
- hPrevInstance is always 0 (it is from the 16-bit days)
- pCmdLine contains the command-line arguments as a Unicode string
- nCmdShow is a flag that says whether the main application window will be minimized/maximized/...

Linux: the address of `main`

`https://refspecs.linuxbase.org/LSB_3.1.0/LSB-generic/LSB-generic/baselib---libc-start-main-.html`:

In Gdb we can ...

- `start`
- `b __libc_start_main`
- `c`
- find out its first parameter

  32 bits: `p *(void **)($esp+4)`

  64 bits: `p $rdi`

Comments

Comments can be added to any instruction/data-item, with ; (semicolon)

Five categories:

End-of-line (EOL) Displayed to the right of the instruction

Pre Displayed above the instruction and in decompiled code

Post Displayed below the instruction

Plate Displayed as a block header above the instruction, and in decompiled view. Plate comments are automatically surrounded by '*'s

Repeatable Displayed to the right of the instruction if there is no EOL comment. These are also displayed at the “from” address of a reference (if there is no EOL or repeatable comment defined at that address)

Constants

- convert
- set equate

Note: decompiler and listing views are not "synchronized" as conversions/equates go

LLMs

LLMs can be leveraged to improve the decompiler output; some notable Ghidra plugins/scripts are:

- GhidrOllama
  `https://github.com/lr-m/GhidrOllama`
  that allows you to use any Ollama Models `https://ollama.com/library` *offline*
- DAILA
  `https://github.com/mahaloz/DAILA`
  AFAIK needs an internet connection to leverage OpenAI's GPT, etc.

Interesting video: *Your Teammate Isn't Human: Mixing Decompilation and AI for Modern Reverse Engineering*
`https://www.youtube.com/watch?v=HbrebQiFLDs`

Calling conventions

Both `function1` and `function2` takes three integers and print a simple calculation.

Let's find out what
→ `function1` and
→ `function2` do by ...

1. running them, and guessing
2. using Ghidra decompiler/function graph

Analysis of restricted-area v2.0

. . . and we're back to restricted-area!

- Can you find the address of `main`?
- Identify the function that prints the flag

Why such a function is not called? Can we do something about it?

Two possibilities are:

- altering the control flow by using a debugger
- patching the machine code

Outline

Dynamic Analysis tools

We can observe the execution of a program at many different levels; e.g.,

Linux

- `strace [-f] [-e ...]`
- `ltrace` — some versions don't work with eagerly-bind executables; check with: `readelf --dynamic ...| grep NOW`

  Try: `ltrace -e strcmp ./restricted_area_v1 # same for v2`
- gdb + GEF

Windows

- Process Monitor from the *Sysinternals Suite*
- Tiny Tracer `https://github.com/hasherezade/tiny_tracer`
- x64dbg

...and even making our custom analyses by leveraging instrumentation frameworks

Introduction

Debuggers are software (or hardware) that permit to get an insight into what a program/system is doing, by

- pausing the execution at specific places
  - optionally, when some conditions are met
- showing the contents of registers and memory
- resuming the execution
- ...

Symbols and debug information

Programs and libraries can include

- Symbols, mapping names to memory addresses. For instance, they allow you to find in which function the *instruction-pointer* is, . . .
- Full debug information, that allow a debugger to match machine code to its corresponding source-level constructs

Released programs typically don't, but they rely on standard libraries. . .

Symbols in Linux

Libc symbols are distributed separately

- in Ubuntu, `libc6-dbg` and `libc6-dbg:i386`
- in gdb (or `~/.gdbinit`) use:
  `set debug-file-directory /usr/lib/debug`
  then,
- `info address` *`symbol`* shows where data for symbol is stored
- `info symbol` *`addr`* prints the name of symbol stored at *addr*

You can also use `addr2line(1)` to read symbol information

Compiling with `-g`/`-ggdb` adds debug information using DWARF [Eag12] (`https://dwarfstd.org/`)

- to dump DWARF information, `dwarfdump`; e.g. `-l` prints the association between PCs and source lines
  → `c-examples/buggy_factorial`

Symbols in Windows

Symbol information can be downloaded from a symbol server

- official one: `https://msdl.microsoft.com/download/symbols`
- you can set the environment value `_NT_SYMBOL_PATH` to something like: `srv*`*`your-cache-path`*`*`*`server-url`*; e.g.
  `setx _NT_SYMBOL_PATH ^`
  `srv*c:\sym*https://msdl.microsoft.com/download/symbols`
- x64dbg set the server in: Options → Preferences → Misc
- `symchk.exe` from Windows SDK, and 3rd-party utilities, can download symbols and store them in the cache

With MS's compiler, `/DEBUG` add debugging information:

1. The linker puts these information into a program database (PDB) file
2. The executable/DLL contains the path of the corresponding PDB
3. A debugger reads the embedded name and uses the PDB

Outline

Experiment

Let's open three times each `restricted_area_v{1,2}` in the debugger.

Obviously, the code is different for v1/2, but you should notice something else…

ASLR

In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries

`https://en.wikipedia.org/wiki/Address_space_layout_randomization`

Memory map

You can synchronize the static and dynamic addresses by setting the Image Base from the Memory Map

- however, there other ways. . .

Useful settings

- You can disable ASLR by

  Linux using the command `setarch`; for instance:

  ```
  setarch $(uname --machine) --addr-no-randomize bash
  ```

  Windows resetting the flag `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` (0x40) in the field `DllCharacteristics` of the `PE_IMAGE_OPTIONAL_HEADER` (e.g., by using PE Bear)

- Windows 10+ implement parallel loading by creating a thread pool of worker threads when the process initializes. You can set registry key

  `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\`*`your-program.exe`*`\MaxLoaderThreads`

  to 1, to avoid having multiple threads around

Restricted area, again!

Let's solve `restricted_area_v2` with the help of a debugger

Demo/exercise

Volatility is a tendency to change quickly, as it happens with the flag of this challenge.
No, this is not a forensics challenge.
Can you be fast enough to catch the flag?

$\rightarrow$ `volatility`

Synchronizing static and dynamic analysis

Two different approaches:

- Ret-Sync, which allows you to use any debugger
- Ghidra "internal" debugger; some shortcomings/limitations at the moment

Ret-sync

- ret-sync stands for *Reverse-Engineering Tools SYNChronization*
- plugins to synchronize a debugging session with a disassembler

`https://github.com/bootleg/ret-sync`

Remote debugging

Ret-sync can be configured via its “.ini” `.sync` file

```
[INTERFACE]
host=...
port=...
```

To enable remote-debugging:

1. find the IP address of the computer running Ghidra
   - typically, Ghidra is run on the host and the debugger inside a VM
   - in VMware, the host-only network (VMnet1) allows you to access an isolated virtual network, completely contained within the host system
2. then, create the configuration file and copy it to both:
   - the user's home directory of the debugger machine `c:\users\`*`user`*`\.sync`
   - the user's home directory or the Ghidra project folder (`.../`*`project`*`.rep/.sync`) of the Ghidra machine

Ret-sync bugs

Sometimes ret-sync doesn't work properly with gdb ☹

The following workaround, in the `.sync` file (inside the Ghidra project folder; i.e., `.../`*`project`*`.rep/.sync`), seems to solve the problems:

- set `use_raw_addr=true` under section `[GENERAL]`

Ret-sync shortcuts in Ghidra

Most important are:

F2	Set breakpoint at cursor address
Ctrl-F2	Set hardware breakpoint at cursor address
Alt-F3	Set one-shot breakpoint at cursor address
Ctrl-F3	Set one-shot hardware breakpoint at cursor address
Alt-F2	Translate (rebase in debugger) current cursor address
F5	Go
F10	Step over
F11	Step into

Code Coverage

- To collect coverage data DrCov, leveraging *DynamoRio*:
  `drrun.exe -t drcov -- program`
  - A similar tool for Intel Pin can be found inside
    `https://github.com/gaasedelen/lighthouse`
    the project that inspired Cartographer and Lightkeeper
- To analyze/visualize, Cartographer: `https://github.com/nccgroup/Cartographer`
  - another, similar, project is: `https://github.com/WorksButNotTested/lightkeeper`

```
UVar2 = (UINT)unaff_RBX;
uVar4 = __scrt_initialize_crt(1);
if ((char)uVar4 == '\0') {
  FUN_140001db0(7);
}
else {
  bVar1 = false;
  uVar9 = 0;
  uVar4 = __scrt_acquire_startup_lock();
  UVar2 = (UINT)CONCAT71((int7)((ulonglong)unaff_RBX >> 8),(char)uVar4);
  if (DAT_140022af0 != 1) {
    if (DAT_140022af0 == 0) {
      DAT_140022af0 = 1;
      uVar5 = FUN_1400089a4((undefined **)&DAT_1400172c8,(undefined **)&DAT_140017300);
      if ((int)uVar5 != 0) {
        return 0xff;
      }
      FUN_140008960((undefined **)&DAT_1400172b0,(undefined **)&DAT_1400172c0);
      DAT_140022af0 = 2;
    }
    else {
      bVar1 = true;
      uVar9 = 1;
    }
```

Pwntools

Pwntools is a

- CTF framework and
- exploit development library

written in Python

https://github.com/Gallopsled/pwntools

Demo/exercise

Demo/exercise:
$\rightarrow$ `bomb` (local and “remote”)

Tubes: `pwnlib.tubes`

- `process`
- `sock`
  - `remote`
  - `listen`
- `ssh`
- ...

various methods to interact:

- `send*[after]`
- `recv*`
  - `clean`
    returns all buffered data from a tube by calling `recv` with a low timeout until it fails
- `interactive`
  prints a prompt, and simultaneously reads & writes

`https://docs.pwntools.com/en/stable/tubes.html#module-pwnlib.tubes`

Context

Many settings controlled via `context`, such as

- `os`: target OS, see `pwnlib.context.ContextType.oses`
- `arch`: architecture; see `pwnlib.context.ContextType.architectures`
- `bits` / `endian`: bit-width/endianness
- `log_level`: logging level; default `logging.INFO`

`context.binary`

The easiest way to *automagically* set all context values is assigning the property `binary`; e.g.:

```
context.binary = './my-binary'
```

(you can also assign an ELF object, which we'll encounter in a few slides)

Packing and unpacking of strings

```
>>> p8(0), p16(0), p32(0), p64(0)
(b'\x00', b'\x00\x00', b'\x00\x00\x00\x00',
 b'\x00\x00\x00\x00\x00\x00\x00\x00')
>>> p32(0xdeadbeef)
b'\xef\xbe\xad\xde'
>>> p32(0xdeadbeef, endian='big')
b'\xde\xad\xbe\xef'
>>> hex(u32(b'\xbe\xba\xfe\xca'))
'0xcafebabe'
```

https://docs.pwntools.com/en/stable/util/packing.html

Endianness

context-aware; can be overridden in the parameters

Magic Command-Line Arguments

Settings, when run in `from pwn import *` mode, can be specified by

- adding UPPERCASE arguments to the command-line; those arguments are extracted, and removed from `sys.argv`
- using environment variables, prefixed by `PWNLIB_`

For instance, to enable more verbose debugging:

```
$ PWNLIB_DEBUG=1 python exploit.py
$ python exploit.py DEBUG
```

Then, for instance, to switch between a local/remote target:

```
if args.REMOTE: # equivalent to: args['REMOTE']
        io = remote('exploitme.com', 4141)
else:
        io = process('./pwnable')
```

See `https://docs.pwntools.com/en/stable/args.html`

Assemble and Disassemble

```
>>> asm('nop')
'\x90'
>>> asm('mov eax, 0xdeadbeef').hex()
'b8efbeadde'
>>> asm('mov eax, 0').hex()
'b800000000'
>>> print(disasm(unhex('6a0258cd80')))
   0:   6a 02                   push   0x2
   2:   58                      pop    eax
   3:   cd 80                   int    0x80
```

ELF parsing and patching

class `ELF` members:

- `sym[bols]` is a *dotdict* of name to address for symbols
  - `prog.symbols['printf']` can be simplified to:
    `prog.symbols.printf`
    `prog.sym.printf`
- `got` is a dotdict of name to address for GOT entries
- `plt` is a dotdict of name to address for PLT entries
  - for an imported function *f*, *elf*`.plt.`*f* `==` *elf*`.symbols.`*f*
- `search(string, writable = False)` → a generator
  search the virtual address space for the specified string
- `(dis)asm` to (dis)assembly using virtual addresses
- `read/write/save`
- ...

To create an ELF from assembly/bytes: `ELF.from_assembly`, and `ELF.from_bytes`

https://docs.pwntools.com/en/stable/elf/elf.html

Interfacing with GDB

Example:

```
p = gdb.debug(args=..., gdbscript=...)
```

gdb is run in a new terminal, executing the `gdbscript`

- if tmux detected, defaults to new pane; if you want to change that: `context.terminal = ['tmux', 'new-window']`

  `context.terminal = ['tmux', 'split-window', '-h']`
- if Gnome terminal doesn't work automatically, you can try:

  `context.terminal = ['gnome-terminal', '--window', '--', 'bash', '-c']`

See `https://docs.pwntools.com/en/stable/gdb.html` and `https://docs.pwntools.com/en/stable/context.html`

And much more. . .

- finding ROP gadgets
- preparing format string payloads
- generating Python scripts (`pwn template ...`)
- . . .

RTFM `https://docs.pwntools.com/en/stable/index.html`

Jupyter notebooks/PyCharm

to use pwntools inside some environments, you need to set the variable `PWNLIB_NOTERM`; e.g.
`PWNLIB_NOTERM=1 jupyter notebook`

Debuggers

https://twitter.com/garabatokid/status/1192753360497197056

Introduction

Instrumentation means inserting new code into a program or a process, to

- observe and/or
- change

its behavior.

- Easy with source programs
  - manually
  - automatically; for instance, in gcc you can use:
    - `-fsanitize=address` to instrument memory access instructions to detect out-of-bounds and use-after-free bugs
    - `-pg`/`-finstrument-functions` to generate profiling code
    - ...
- Interesting on (stripped) binaries ☺

Scenarios

Let's consider three scenarios:

1. Replacing existing code/AKA "patching"
    - This can also be seen as "removing", since we can overwrite with NOPs
2. Inserting/removing code in the middle of existing one
3. Adding new code "somewhere"

Replacing existing code

It's *relatively* easy to replace existing instructions, as long as

1. the new code fits into the space of the old one
2. jump targets are preserved

Examples of safe transformations are:

- "NOPping" annoying checks (anti-analysis? ☺)
- transforming conditional jumps to unconditional ones
- changing constant operands
- ...

e9patch [DGR20] relaxes the former constraint by "punning" bytes following an instruction. I.e., those bytes are used both as the last bytes of the inserted jump *and* the bytes for following instructions

	mov %rax,(%rbx)						
Original:	48	89	03	48	83	c0	20
Patched:	e9	XX	XX	48	83	c0	20
	jmpq 0x8348XXXX *(punned)*						

Inserting/removing code

Inserting new (or removing old) code is extremely hard, quoting [And18]:

> ... *new code will shift existing code to different addresses, thereby breaking references to that code. It's practically impossible to locate and patch all existing references after moving* ...

So, manually patching it's possible, but *tough* ... interesting read:
Did Microsoft just manually patch their equation editor executable? Why yes, yes they did. (CVE-2017-11882)

Instrumentation framework

For more complex scenarios, there are Instrumentation frameworks

- tools for building program analysis tools
- they allow you to choose what to instrument and how

We can't cover them now; the most prominent are Frida `https://www.frida.re/`:

- "Unlocking secrets of proprietary software using Frida" @ NDC 2018
  `https://www.youtube.com/watch?v=QC2jQI7GLus`
- "The engineering behind the reverse engineering" @ OSDC 2015
  `https://www.youtube.com/watch?v=uc1mbN9EJKQ`

and Intel's Pin `https://www.intel.com/software/pintool`

Restricted area, again *and again*!

Let's solve `restricted_area_v2` by patching the executable

Demo/exercise: restricted-area v3.0

We promise this is the last variant!
→ `restricted_area_v3`

... you know what to do, don't you? ☺

Start the server by running:
`restricted_area_v3_run_server.sh`
The server listen at port 6001/tcp, that can be reached by running:
`nc 127.0.0.1 6001`

Outline

Demo/exercises

Exercises:

→ `math_is_4_fun`

→ `minions`, port 6002

→ `jurassic_park`

→ `slow_printer`

→ `easter_egg`

→ `lucky-numb3rs`, port 6003

→ `the_maze`, port 6004

→ `unbreakable_aes`

Demo/exercise:

→ `pacman4console`

References I

[ACvdV+16] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos.
An in-depth analysis of disassembly on full-scale x86/x64 binaries.
In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, Austin, TX, 2016. USENIX Association.

[And18] Dennis Andriesse.
Practical Binary Analysis.
No Starch Press, 2018.

[ASB17] Dennis Andriesse, Asia Slowinska, and Herbert Bos.
Compiler-agnostic function detection in binaries.
In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 177–189. IEEE, 2017.

References II

[DGR20] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury.
Binary Rewriting without Control Flow Recovery.
In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[Eag12] Michael J Eager.
Introduction to the DWARF debugging format, 2012.

[JLH13] Christopher Jamthagen, Patrik Lantz, and Martin Hell.
A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries.
In *Anti-malware Testing Research (WATeR), 2013 Workshop on*, pages 1–9. IEEE, 2013.

References III

[JRM11] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller.
Labeling library functions in stripped binaries.
In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.

[JZL+20] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren.
An empirical study on ARM disassembly tools.
In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 401–414, 2020.

[LD03] Cullen Linn and Saumya Debray.
Obfuscation of executable code to improve resistance to static disassembly.
In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

References IV

[MM16] Xiaozhu Meng and Barton P Miller.
Binary code is not easy.
In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35. ACM, 2016.