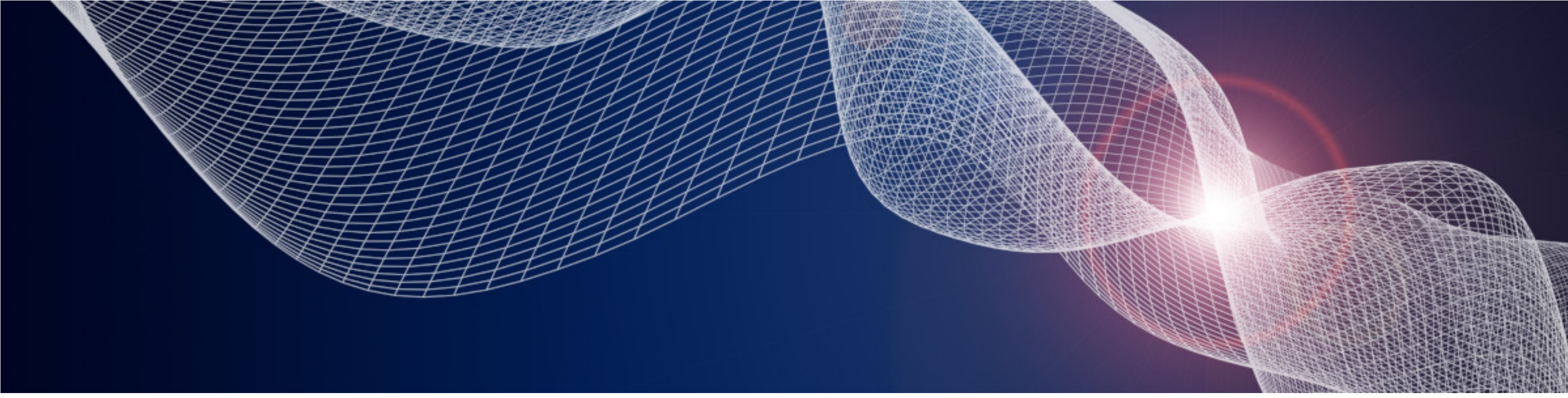
Table of Contents



1. Introduction

Welcome to the concepts book of the *Assignment* **Reverse Engineering**. The chapters of this assignment will teach you the skills required to follow the different lectures and to solve the exercises. They will introduce you to three main techniques which are key to reverse engineering code:

- Deobfuscation
- Static analysis
- Dynamic analysis

1.1 Techniques

To get started, let's have a quick look at what these words / expressions actually mean.

1.1.1 Deobfuscation

Deobfuscating program code means turning (intentionally or unintentionally) confusing and hard-to-read instructions into code which is easily understood by a human.

1.1.2 Static Analysis

Statically analyzing a program means understanding what it does (and possibly how) without executing it. Static analysis is naturally the safer approach when dealing with code we don't yet understand; and it's definitely safer if we know it's malicious.

1.1.3 Dynamic Analysis

Unfortunately, it's not always possible - or at least not humanly possible - to get to a result by statically analyzing a program. If that is the case, we can dynamically analyze it by using different debugging techniques. If possible, we will want to avoid running such programs on our own machines and execute them in virtual machines, emulators or simulators.

1.2 Feedback

Please let me know of any issues you find with the chapters that make up this course. It's very well possible that

- I got something wrong or missed information I should be giving you
- The explanations could be better - either in general or for your personal way of learning

Even if it turns out that you misunderstood something, we get two positive results:

- You get to learn something
- Future students get improved scripts

Also, please let me know if you later discover something you would have liked me to have covered. I will be happy to give you access to updated versions of this script if you're interested.

1.2.1 Breakdown

- $1/4$ will be theory - teaching you the relevant concepts.
- $3/4$ will be practice - having you apply the relevant methods.

1.3 Structure

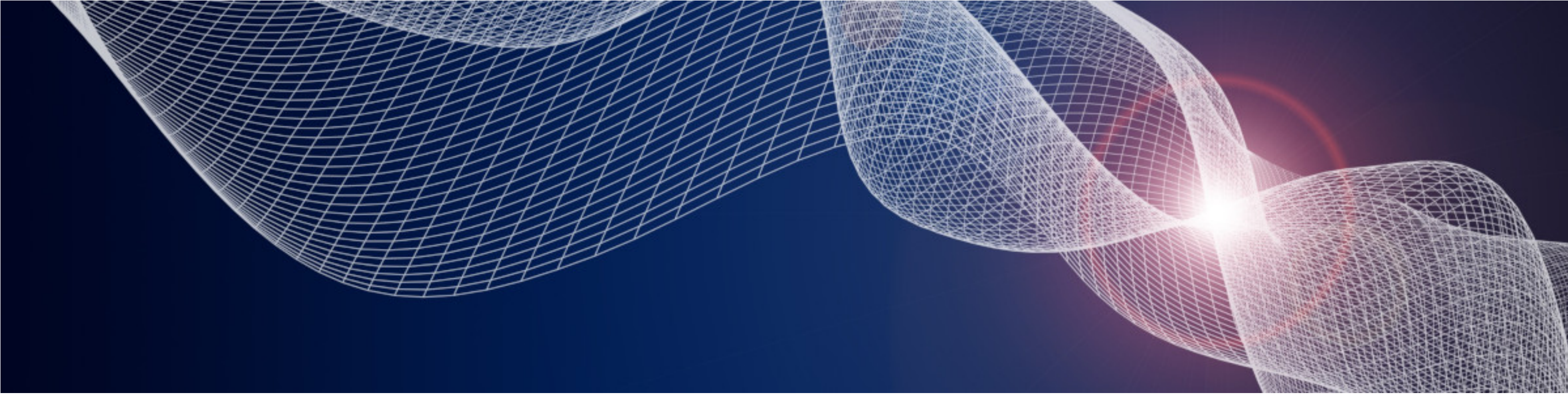
What you're holding in your hands is the initial instalment of the concept book. You will get each chapter's content in two versions to choose from:

- an on-screen version
- a printable booklet, which you can add to the cover to create a complete book

There are two more books:

- one containing the exercises that accompany the concepts and which you're very much invited to complete.
- one containing possible solutions to the exercises - you will receive these over time.

> In case you get access to earlier versions of the "Solutions" book, you may be tempted to look at these before coming up with your own solutions. ***Don't*** - you'd only be putting yourself at a disadvantage: solving the exercises yourself at first is how you get an understanding of how much of the material you've mastered so far and where you may wish to invest more time; and above all, you get better at it over time.



2. Deobfuscation

2.1 Introduction

These days, most programs you get your hands on will have been transformed from the original source code in some way. In most types of programmes, these transformations are not specifically aimed at making the inner workings harder to understand, but serve purposes like minimizing the amount of data transferred via networks (allowing the programs to load faster), optimizing performance or simply making the program executable by a machine.

Some programs, though, are transformed with the express goal of making them harder to understand (a prime example of this of course is malware). These transformations are called *code obfuscation* [1] .

*De*obfuscation then is the art of turning a such transformed program back into source code that can be read, understood, and possibly patched to do things the original programmer didn't intend.

We will first discuss different obfuscation methods, then how to counter them and finally look at tools which can help us deobfuscating code in three different languages: VBA, PowerShell, and JavaScript.

Why these? Well, these languages are often involved in today's attack vectors, e.g. when targeting a windows PC (VBA as entry door via office documents, PowerShell to interact with the outside world) or when attacking unsuspicious visitors of infected or malicious websites.

2.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Obfuscation	60		60
Analysis	60	360	420
Total	**120**	**360**	**480**

[1] https://en.wikipedia.org/wiki/Obfuscation_%28software%29

2.3 Obfuscation

Good programs are written in a way that makes them easily readable and thus maintainable. How is that achieved? There are a some high level rules to adhere to:

- Choose good names: names which make explanations unnecessary. As an example, a method is probably better named "findNonAdminUsers" than "load".
- Divide code up in small, self-contained methods. A method loading values from a database, calculating metrics, creating a visualization and exporting that along with raw data to PDF should probably be divided up into different methods doing each step and one top method orchestrating these.
- Group together what belongs together to make it easier to find the next piece of code to read in a control flow.
- Code should be as short as possible (but the above rules take precedence) and not contain redundancy.

In order to understand how to deobfuscate code, we should first understand some techniques commonly used to obfuscate code. What does obfuscation do? It turns well-written - and thus easily readable and understood - programs into very unreadable ones. This is typically done by inverting the rules listed above ;-)

2.3.1 Name Mangling

One of the things that help us most understand code is good naming of variables and methods. So the easiest way of confusing human readers is to arbitrarily change these names to something completely random or even intentionally confusing.

> Name mangling is not always malicious: JavaScript minification shortens all possible names as much as possible as a form of compression which allows to save bandwidth when transferring code between server and clients viewing web content in their browser.

Let's have a look at an example.

Original

```
method calculateFactorial = (input) => {
  variable result = 1
  variable counter = 1

  loop
    result = result * counter
    exit if counter == input
    counter = counter + 1
  end loop
}
```

Mangled

```
method ydfjkwe823 = (ydfjkwe822) => {
  variable ydfjkwe821 = 1
  variable ydfjkwe820 = 1

  loop
    ydfjkwe821 = ydfjkwe821 * ydfjkwe820
    exit if ydfjkwe820 == ydfjkwe822
    ydfjkwe820 = ydfjkwe820 + 1
  end loop
}
```

The two code snippets do exactly the same - would you agree the original one can more easily be understood?

2.3.2 Data Transformation & Distribution

Even if all variables are renamed, the values assigned to them may still reveal much of what they are meant for. This is even more true for strings which may be output to the user such as prompts to input a value. In order to hide these from human readers, one good approach is to put transformed values into the code and to apply the reverse transformation at runtime. Even legitimate code may have good reasons to hide such strings (e.g. for simple license keys).

Here are some different ways of transforming strings:

- decomposition (e.g. `"We" + "lc" + "om" + "e!"`)
  - String formatting can be used to recompose strings in different orders (e.g. `"{2}{0}{1}"-f "i","!","H"` in * *PowerShell*)
  - Any kinds of operations can be used to compose data (e.g. `5**3//20%5` instead of just `1` in *Python*)
- transforming them letter by letter
  - simply presenting them in e.g. hex (e.g. `"\x4f\x68\x20\x6d\x79\x21"` in *Python*)
  - using maths to represent ASCII codes (e.g. `chr(87)+chr(101)+chr(105)+chr(114)+chr(100)+chr(33)` in *JS*)
  - leveraging language features (e.g. `(![]+[])[!+[]+!+[]]+([![]]+[][[]])[+!+[]+[+[]]] +(![]+[])[+[]]+(!![]+[])[(+!+[]<<+!+[])+!+[]]` in *JS*)
- adding unnecessary escape characters (e.g. `"di^r c^:^\"` in *PowerShell*)
- embedding fixed patterns which will first have to be removed (e.g. `"Ha8]ea8]la8]la8]oa8]!"`)
- encoding them (e.g. in base64: `"U2VlPwo="`)
- encrypting them (e.g. `"oi93ssdf2eo23"` - don't waste your time on this ;-))

To add to this, a further complication is to assign different parts of the data to different variables all over the place and to reconstruct them as needed.

2.3.3 Code Bloating

One quite effective deterrent to deobfuscation can be inserting code which doesn't change anything, but greatly increases the amount of code the viewer has to look at. Some data transformation techniques have this effect as well (e.g. letter by letter transformations, see above), so they're often used in combination with pure code bloating techniques.

There are two basic methods used to bloat code:

- Inserting dead code
- Adding identity operations - operations which don't do anything

Both techniques have in common that portions of the code can be removed without any effect on the result.

2.3.3.1 Dead Code Insertion

One very simple and effective way to make code more complex to read is to add control structures where none are required. As a very simple example, we can add code within a conditional block which will never execute:

JavaScript

```javascript
if (!(""+[]) && "false" != !true) {
  console.log("this will run - make sure you understand why!");
} else {
  console.log("this will never run");
}
```

The same concept can of course be used with loops (which run e.g. zero times, are exited early, ...) and case matching (where only one or even no case matches). There is one additional concept we have to be aware of: `GoTo $label` to jump to a position in code which is labelled `$label`. `GoTo` is not typically taught to programmers these days because it's considered bad style (it's very hard to keep track of what's happening), but it's very effective for exactly that reason!

VBA

```vba
sub foo()
  GoTo aweridfjlk
  ' bloat code
  ' more bloat code
aweridfjlk:
  Debug.Print "this actually happens"
  GoTo weorilksjdf
  ' this doesn't
  ' nothing here to see...
weorilksjdf:
end sub
```

2.3.3.2 Identity Operations

You may remember from maths that there are so-called *identities* for different types of operations. As an example, the *additive identity* for natural numbers is `0`, and their *multiplicative identity* is `1`. The same concept can be applied to different data types, allowing us to pretend to be doing complicated things while not doing anything at all. One common pattern we see is declaring uninitialized variables (or using implicitly declared variables) and concatenating them to strings:

VBA

```
asdfle = weofvd + "ll" + dlfier
' ...
werkdr = asasfe + "He" + werlkf
' ...
sdflgkj = owersd + werkdr + flojse + asdfle + asdfke + "o"
```

2.3.4 Control Flow Rerouting

If we want code to be hard to read, we should make sure we make readers jump around as much as possible to keep them from getting a clear picture of what's happening. Two efficient ways of doing that are

- *inlining* code: copying code from method implementations and replacing calls to those methods by pasting that entire code (with the parameter names replaced by the values calculated in the caller method)
- *flattening* control flow by creating a “dispatcher” (usually in the form of case matching), which executes different bits of code based on a variable which represents the next “label” to go to.

Original

```
$a=$args[0]




if($a -eq 0) {







  write-host 1
} else {


  write-host 10



}
```

Obfuscated

```
$a=$args[0]
$b = 0;
while(1) {
  switch($b) {
    0 {
      if($a -eq 0) {
        $b = 1
      } else {
        $b = 2
      }
      break;
    }
    1 {
      write-host 1
      return;
    }
    2 {
      write-host 10
      return;
    }
  }
}
```

2.3.5 Dynamically Executed Code

Many languages can execute code contained in strings. This allows creating code within running code and then executing it, making it harder to understand what's going on in the end. Here are two examples of how code can be run:

PowerShell

```powershell
$c=$ExecutionContext.InvokeCommand.NewScriptBlock("write-host Hi!")
Invoke-Command $c
```

JavaScript

```javascript
eval("console.log(2 + 2)")
```

2.4 Analysis

If possible, use a tool to deobfuscate code you encounter. There are many of them around (we will look at a few later), and they can usually help you at least with certain parts of the deobfuscation. Since tools are usually dumb, though, we should be able to do this manually as well so we can deal with situations which were not foreseen by the tools' authors.

2.4.1 General

Modern IDEs such as Visual Studio Code [2] (VSCode) give us a lot of help when deobfuscating code, for example by showing us unused variables or allowing us to do rename refactorings. If you haven't already, please download and install the application in your working environment.

One thing I would definitely recommend is saving intermediate steps we come up with. The reason is simple: if we make a mistake and e.g. remove code that later turns out not to be "dead", but is dynamically called, we may have to redo many of the steps we had already completed for that "resurrected" piece of code.

A very simple way of doing that is to initialize a git repository in the folder you do your work in. If you're not familiar with Git, here's [3] a very short intro. Once you've got a git shell installed on your system, you can follow these steps (in your working directory):

1. `git init`
2. `git add .`
3. `git commit -m "Initial"`
4. Peel away an obfuscation layer
5. `git commit -m "some message"`
6. Not yet done? Go to (4)

You can also implicitly save intermediate steps by using e.g. the Local History [4] extension for *VS Code*.

An alternative to that approach is to script the deobfuscation steps, e.g. in Python - or using specific tools. One very well-known general-purpose tool called *The Cyber Swiss Army Knife* is CyberChef [5] . It lets you create "recipes", which allows you to daisy-chain operations transforming inputs. You can either use the tool directly online, download it to your machine or use one of the available docker images (e.g. this one here [6]).

[2] https://code.visualstudio.com/
[3] https://guides.github.com/introduction/git-handbook/
[4] https://marketplace.visualstudio.com/items?itemName=xyz.local-history
[5] https://gchq.github.io/CyberChef/
[6] https://github.com/mpepping/docker-cyberchef/

2.4.1.1 Steps

There's not a generic approach to deobfuscating that will be the most straight forward in every case, but we can generally say that if we aim to quickly reduce the amount of code we're dealing with, we will likely save a lot of time. I would recommend the following basic "cookbook":

1. Run a code formatter
2. Unbloat
    1. Remove dead code
    2. Remove identity operations
3. Compact & clean up data
    1. Evaluate constant mathematical expressions
    2. Apply constant string concatenations
    3. Inline data from other sources
    4. Decode / decrypt

You may have to do certain steps more than once if a later transformation re-introduces elements that were cleaned up previously (e.g. if dynamic code is introduced which again has been obfuscated).

At any point, it can help to rename method names or variables which we recognize for what they are. Use your IDE's refactoring capabilities to do this!

2.4.2 VBA

The first step to understanding VBA code inside office documents is to get access to the code - without exposing ourselves to the risk of executing malicious code. As always, there are lots of ways of doing so - we will use a tool written in Python to do it: olevba [7], which is part of the oletools [8]) suite.

The official installation instructions ask you to execute `sudo -H pip install -U oletools`, but that seems a bit risky, so we'll take the safer path:

1. Make sure you have pipenv installed [9] (`which pipenv`)
   - You may also have to install `pip` [10] first.
2. Create a separate folder somewhere (e.g. `~/reversing/oletools`) and `cd` into it
3. Run `pipenv install oletools` - this will install oletools into a user specific virtual environment

With this out of the way, from the folder we created, we can now run the different tools in the *oletools* suite, e.g. `pipenv run olevba $\$$FILE` to extract VBA code from an office document.

Time to exercise!

2.4.2.1 Automated Deobfuscation

There currently doesn't seem to be much of an alternative to ViperMonkey [11]. The program implements its own VBA parser in Python and is pretty successful at deobfuscating code. The downside is that of course the more dedicated malware authors know that the parser is not complete and thus find ways to crash it.

To use it, simply clone or download [12] the repository, `cd` into the main directory and run `./docker/dockermonkey.sh $maldoc`, with `$maldoc` being the document you want to deobfuscate.

The nice thing about this solution is that `dockermonkey.sh` automatically disconnects the docker container's network, thus effectively creating an isolated sandbox for analysis.

> `ViperMonkey` uses `olevba` in the background, so no need to extract the code manually.

Time to exercise!

[7] https://github.com/decalage2/oletools/wiki/olevba
[8] https://github.com/decalage2/oletools
[9] https://pipenv.pypa.io/en/latest/install/#installing-pipenv
[10] https://pip.pypa.io/en/stable/installation/
[11] https://github.com/kirk-sayre-work/ViperMonkey
[12] https://github.com/kirk-sayre-work/ViperMonkey/archive/refs/heads/master.zip

2.4.3 PowerShell

When looking at obfuscated PowerShell code, you're most likely not dealing with something created by a benevolent programmer, so I would recommend you do any work in a virtual machine. You can get a Windows VM from here [13] .

2.4.3.1 Language

If you don't know *PowerShell*, but have mastered *Bash* syntax, you're in luck: the two are rather similar, so if you can read bash syntax, there's no reason you should struggle with *PowerShell*.

Let's look at some differences between the languages so you can hit the ground running.

Area	Bash	PowerShell	Comment
String comparison	`=`, `!=`	`-eq`, `-ne`	
Logical operators	`-o`, `-a`	`-or`, `-and`	
Negation	`! $expr`	`! ($expr)`	
Empty	`-z $var`	`$var -eq $null`	
File existence	`-e $file`	`Test-Path $file`	
Escaping	`\`	`` ` ``	Backtick
Printing	`echo $text`	`Write-Host $text`	
Variable assignment	`foo="bar"`	`Set-Variable foo "bar"`	Also `set foo "bar"` or simply `$foo="bar"`
Path	`$PATH`	`$env:path`	

Control Structures read almost the same, but the keywords surrounding the code blocks are replaced by braces, e.g.

Bash

```bash
if [ condition ]; then
  # ...
elif [ condition ]; then
  # ...
else
  # ...
fi
```

PowerShell

```powershell
if (condition) {
  # ...
} elseif (condition) {
  # ...
} else {
  # ...
}
```

One useful feature you'll likely find "abused" in obfuscated code is string formatting using "-f", described e.g. here [14] . Here's an example:

PowerShell

```powershell
"{1}, {0}" -f "there!","Hi" # => Hi, there!
```

Time to exercise!

[13] https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/
[14] https://ss64.com/ps/syntax-f-operator.html

2.4.3.2 Automated Deobfuscation

A tool which is rather nice is PSDecode [15] - I've created a fork [16] to help us a bit further along with the deobfuscation of the sample I chose. You can follow the simple installation instructions in the `README.md` file to set it up.

This time, I highly recommend you use a VM (or a windows docker instance if you are working on Windows) and to disconnect all access to your host machine when running the tool.

Time to exercise!

[15] https://github.com/R3MRUM/PSDecode
[16] https://github.com/chezwicker/PSDecode

2.4.4 JavaScript

JavaScript often serves to deliver malware as well, often via attachments to email opened by unsuspecting recipients. Like VBA, the code executed very often downloads the actual payload and executes it via *PowerShell*, targeting *Windows* environments.

A common entry point to the program delivered is an *immediately invoked function expression*. This works by defining a function immediately followed by the arguments it will be passed, e.g.

JavaScript

```javascript
(function x(a) {
  console.log(a);
})(1)
```

Two further features of JavaScript which are very important in (de)obfuscation are property access by name and the comma operator. Let's make sure we understand these!

- In an object `x = { foo: 1, bar: true }` we can access property `bar` by using `x["bar"]`. What may not be completely obvious at first is that this also works for methods. As an example, we an access the method `push` of an array `y` be using `y["push"](2 )` to append the value `2` to `y`. And since we've seen that there are lots of ways to obfuscate strings, this is a great opportunity for control flow obfuscation.
- The comma operator can be used in an expression to evaluate each operand from left to right and to return the result of the last one.

JavaScript

```javascript
x = (console.log("Hi!"), "I'm Bob");
console.log(x);
```

And maybe one more thing of importance that wouldn't be used in clean code: one can declare (and assign) multiple variables following one `var`, `let`, or `const` keyword. The following two code blocks are equivalent

One-Line

```javascript
var x, y = 2, z = (1,2,3);
```

Multi-Line

```javascript
var x
var y = 2
var z = (1,2,3);
```

Time to exercise!

2.4.4.1 Tooling

There's particularly one tool which is useful and actively maintained: box-js [17] . It rewrites the source code to make it easier to read and, importantly, executes it in an emulated Windows JScript environment.

You can also run the analysis in a *Docker* container using the following command, with

- `$hostLocalDir` the path to the directory on your host (VM) containing the JavaScript file(s) to analyze - this is mounted into the docker container at location `/samples`.
- `$dirOrFile` the directory or file to analyze. This will be relative to `/samples`, which is the folder within the container to which the `$hostLocalDir` is mapped.

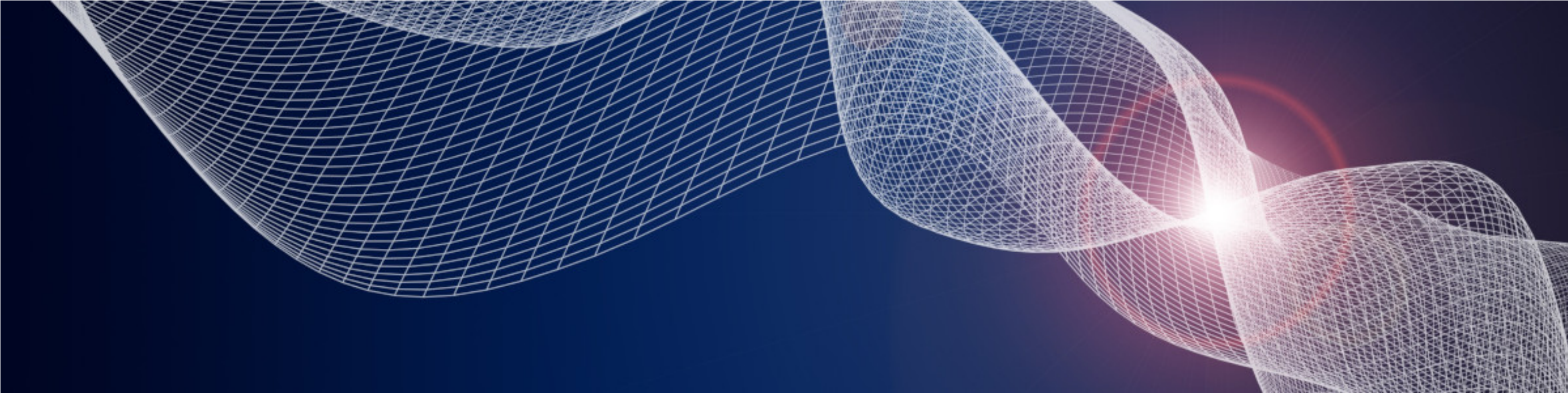
Bash

```bash
docker run \\
  --rm \\
  --volume $hostLocalDir:/samples \\
  capacitorset/box-js \\
  box-js $dirOrFile \\
  --output-dir=/samples \\
  --loglevel=debug
```

When mounting a volume, note that relative paths don't work - `$hostLocalDir` has to be an absolute path. You can use variables like e.g. `$PWD`, though.

Time to exercise!

[17] https://github.com/CapacitorSet/box-js



3. Static Analysis

3.1 Introduction

Static analysis of program code is about understanding what will happen when the code is executed. This is particularly relevant when we don't want to or cannot just execute the code to see what it does (e.g. because it reacts differently when in a sandbox or being debugged) or because we're interested in more than one path the program might take.

When we're dealing with interpreted languages, we rather quickly get to code we can interpret. This is more difficult with compiled languages. We will have a look at disassembling programs and take a deep dive into Ghidra [1] , an excellent software reverse engineering suite developed by the NSA.

In some cases, we will be able to look at decompiled code, but sometimes, we will have to directly read assembly code. Unfortunately, this isn't the easiest material to work through, but we will make sure to cover the basics.

3.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Basics	30	30	60
Ghidra: Installation	15	15	30
Ghidra: Introduction	30	15	45
Ghidra: Component Overview	45	105	150
Ghidra: Program Analysis	45	150	195
Total	**165**	**315**	**480**

[1] https://ghidra-sre.org/

3.3 Basics

Let's start with the basics: gathering information about the executable file we're working with. Even though we will be able to find all of this functionality right within *Ghidra* as well later, the following are basic tools we should always have around and know how to use for a quick first glance.

3.3.1 File Hash

If we don't know what we're dealing with at all, the very first thing we should do is check whether the file we're looking at is a known malware and might have been analyzed by someone else. If so, we can save the trouble of doing everything ourselves.

Most online sites use a file's hash code [2] as an identifier, so we'll want to have that computed first.

On Unix, we can use `shasum -a 256 $path`, leading to the following output (with `/mnt/x/WhatAmI.exe` passed as `$path`):

Output

```
cc4bd8ca652f9ee046518626850c67773a49dead70d9a648987e64cc1313f71a  /mnt/x/WhatAmI.exe
```

On Windows, `Get-FileHash $path` does the trick:

Output

```
Algorithm Hash                                                             Path
--------- ----                                                             ----
SHA256    CC4BD8CA652F9EE046518626850C67773A49DEAD70D9A648987E64CC1313F71A
    x:\WhatAmI.exe
```

There are tons of sites to which hashes can be submitted, but I'd like to mention two in particular:

- VirusTotal [3] is probably one of the most well known sites.
- MalwareBazaar [4] is a relatively recent repository which has quickly gained a lot of traction.

Time to exercise!

[2] https://en.wikipedia.org/wiki/Hash_function
[3] https://www.virustotal.com/gui/home/search
[4] https://bazaar.abuse.ch/browse/

3.3.2 File Type

We will want to figure out what kind of file we're going to analyze. If we're on a *Unix* system, we can simply call `file $path` with the file we're interested in as an argument, for example:

Output of `file WhatAmI.exe`

```
WhatAmI.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

If we're not on *Unix*, we can use the library `python-magic` to get the same result. To use it, first follow the installation instructions here[5] .

With `magic`, we can get the same information `file` provides on *Unix*.

Python

```python
import magic

magic.from_file("WhatAmI.exe")
```

Output

```
'PE32 executable (GUI) Intel 80386, for MS Windows'
```

On Windows, you might have to follow these steps to get everything ready first:

1. Install *Python*. In newer systems, simply typing `python` in *PowerShell* will open the *Windows Store* and offer to install *Python* if you don't have it yet.
2. Optionally upgrade `pip`: `python.exe -m pip install --upgrade pip`
3. Open a new instance of *PowerShell* and type `pip install --user pipenv`.
4. Update your path so you don't always have to specify the full path to `pipenv`:
   ```
   [Environment]::SetEnvironmentVariable("Path",
   ↪ [Environment]::GetEnvironmentVariable("Path",
   ↪ [EnvironmentVariableTarget]::User) +
   ↪ ";$(python -m site --user-site)\..\Scripts",
   ↪ [EnvironmentVariableTarget]::User)
   ```

3.3.3 Strings

Sometimes we can get a lot of information from the static strings contained in a program - that's why (as we saw in the "Deobfuscation" chapter) strings are often transformed in various ways by those who want to keep us from understanding a program.

Nonetheless, we should always see if we can get some information. To extract anything that could be a string (which is everything that consists of a minimum number of printable characters), in *Unix* we can run `strings $path`, e.g. `strings /mnt/x/WhatAmI.exe`. The strings found will be printed line by line. Adding the `-t x` option will prefix the string found with the memory offset (in hex) where it was found in the file.

To get the same functionality in *Windows*, a similar program can be downloaded from here[6] .

[5] https://pypi.org/project/python-magic/
[6] https://docs.microsoft.com/en-us/sysinternals/downloads/strings

3.4 Ghidra

So far, we've only looked at the very surface of the program we're interested in. We know what platform it is targeting and by looking at the contained strings we have gotten a glimpse of how the program interacts with its users. This gives us valuable information, but we will really have to dig into the program to see what it is doing.

In order to do that, we'll have a look at a freely available tool, *Ghidra*. Before it became available to the public, we would have looked at individual disassemblers and decompilers or might have worked with evaluation versions of commercial tools like *IDA Pro*. But fortunately (and maybe somewhat surprisingly at first), the US *National Security Agency* (NSA) was fed up with training their new joiners to use their own reverse engineering suite and have thus decided to open source the very powerful Ghidra SRE [7] (SRE stands for *S*oftware *R*everse *E*ngineering).

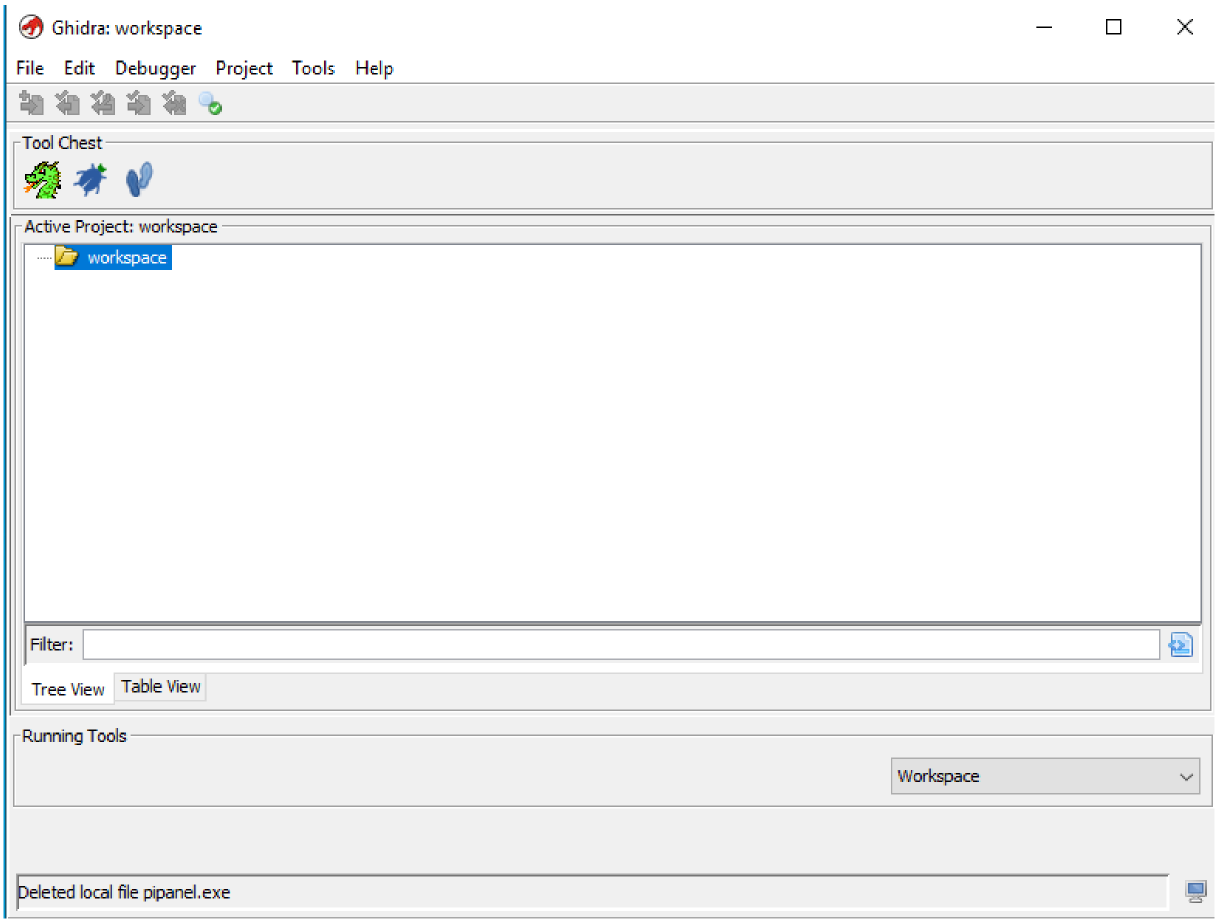
3.4.1 Installation

Download and install the latest release of the software - see detailed instructions in the exercise book.

[7] `https://ghidra-sre.org/`

3.4.2 Introduction

After launching *Ghidra* for the first time, we need to create a project. Go to “File” > “New Project” and choose “Non-Shared Project” to create a local workspace. Now we’re ready to actually do some work!

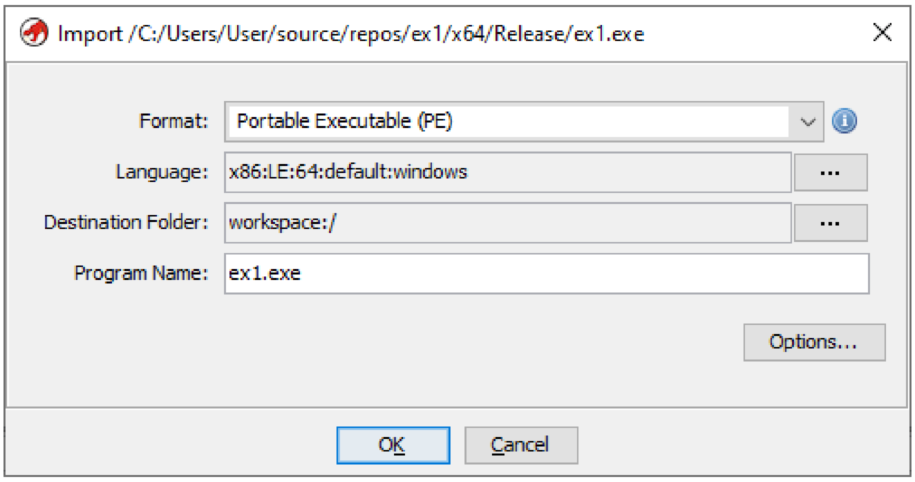


Deleted local file pipanel.exe

Figure 3.1: Initial Screen

3.4.2.1 Importing Programs

To import a program to analyze, go to “File” > “Import File. . . ” (or simply press the key `i`). Choose the program to import and you’ll be presented with a dialog showing basic information about the file.


Import /C:/Users/User/source/repos/ex1/x64/Release/ex1.exe

Format: Portable Executable (PE)
Language: x86:LE:64:default:windows
Destination Folder: workspace:/
Program Name: ex1.exe

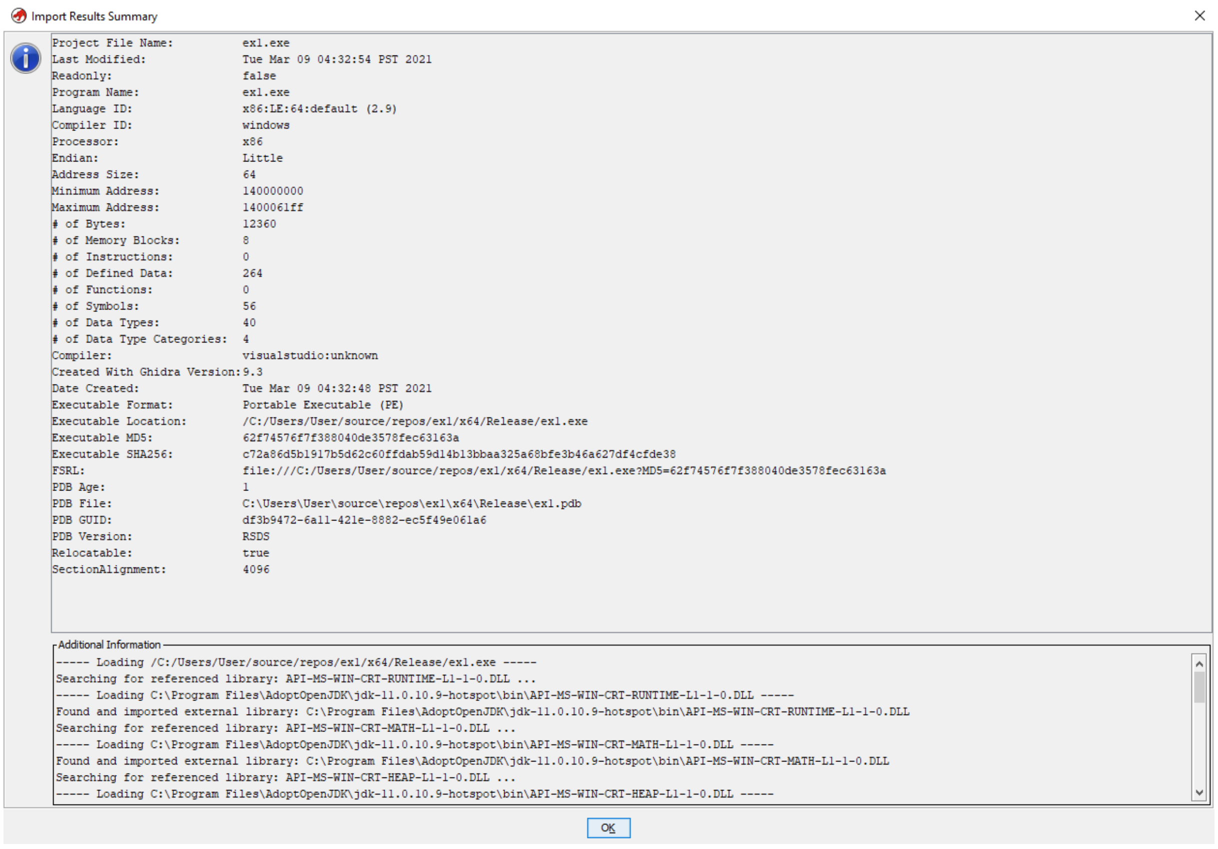
Figure 3.2: Import Dialog

This is what the dialog immediately tells us for the example imported:

- **Format**: the type of file (the file format) is “Portable Executable (PE)”. This is the standard format for windows applications.
- **Language**: the executable . . .
  - **x86**: . . . is aimed at Intel x86 processors.
  - **LE**: . . . has Little Endian [8] memory format.
  - **64**: . . . targets 64bit systems.
  - **default**: . . . no special processor variant.
  - **windows**: . . . was compiled by *Visual Studio* under *Windows*.
- **Destination Folder**: allows to create a structure within the workspace
- **Program Name**: the name of the application being imported.

[8] https://en.wikipedia.org/wiki/Endianness

Once we confirm, the program is imported and we get a much more detailed listing of what *Ghidra* found out about the program.



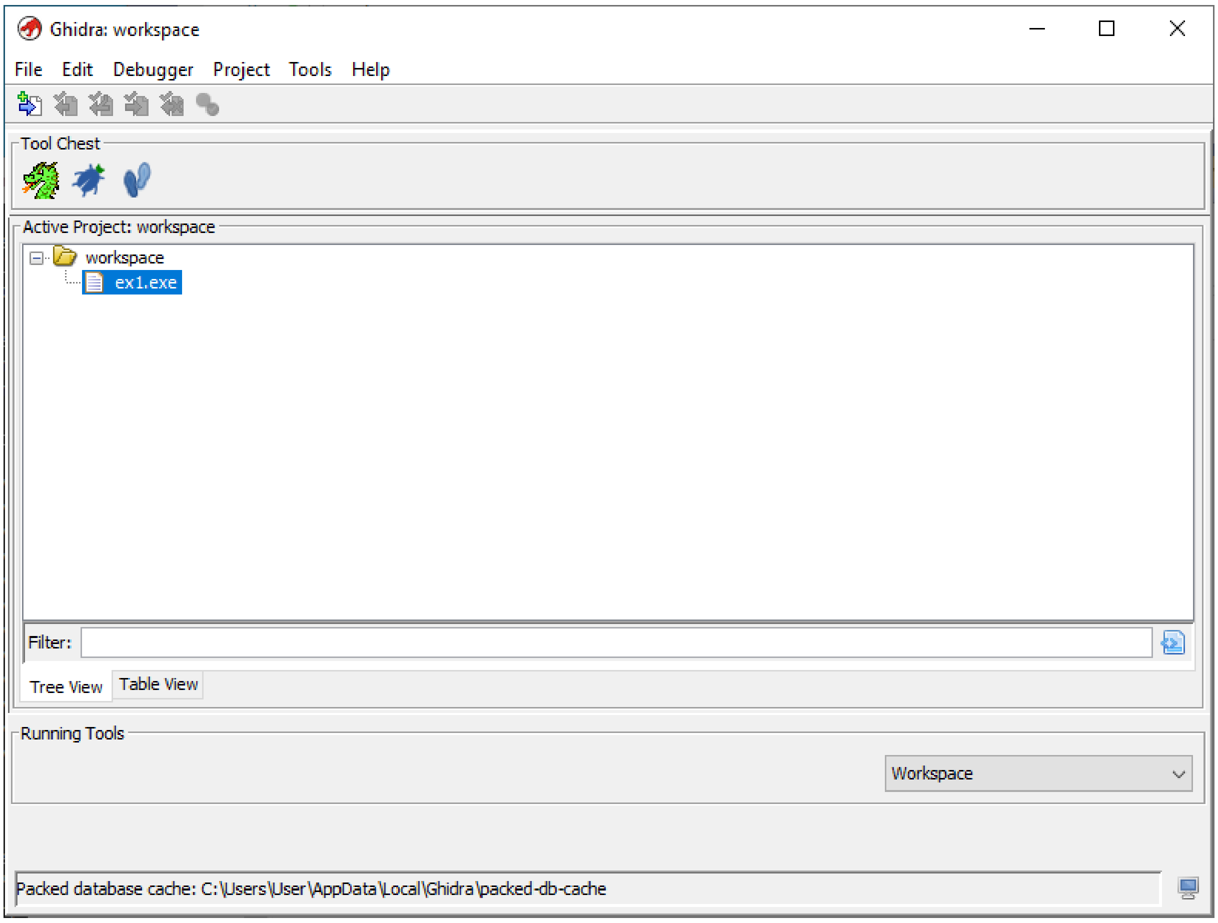
```
Project File Name:          exl.exe
Last Modified:              Tue Mar 09 04:32:54 PST 2021
Readonly:                   false
Program Name:               exl.exe
Language ID:                x86:LE:64:default (2.9)
Compiler ID:                windows
Processor:                  x86
Endian:                     Little
Address Size:               64
Minimum Address:            140000000
Maximum Address:            1400061ff
# of Bytes:                 12360
# of Memory Blocks:         8
# of Instructions:          0
# of Defined Data:          264
# of Functions:             0
# of Symbols:               56
# of Data Types:            40
# of Data Type Categories:  4
Compiler:                   visualstudio:unknown
Created With Ghidra Version:9.3
Date Created:               Tue Mar 09 04:32:48 PST 2021
Executable Format:          Portable Executable (PE)
Executable Location:        /C:/Users/User/source/repos/exl/x64/Release/exl.exe
Executable MD5:             62f74576f7f388040de3578fec63163a
Executable SHA256:          c72a86d5b1917b5d62c60ffdab59d14b13bbaa325a68bfe3b46a627df4cfde38
FSRL:                       file:///C:/Users/User/source/repos/exl/x64/Release/exl.exe?MD5=62f74576f7f388040de3578fec63163a
PDB Age:                    1
PDB File:                   C:\Users\User\source\repos\exl\x64\Release\exl.pdb
PDB GUID:                   df3b9472-6a11-421e-8882-ec5f49e061a6
PDB Version:                RSDS
Relocatable:                true
SectionAlignment:           4096
```

```
----- Loading /C:/Users/User/source/repos/exl/x64/Release/exl.exe -----
Searching for referenced library: API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL ...
----- Loading C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL -----
Found and imported external library: C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\API-MS-WIN-CRT-RUNTIME-L1-1-0.DLL
Searching for referenced library: API-MS-WIN-CRT-MATH-L1-1-0.DLL ...
----- Loading C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\API-MS-WIN-CRT-MATH-L1-1-0.DLL -----
Found and imported external library: C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\API-MS-WIN-CRT-MATH-L1-1-0.DLL
Searching for referenced library: API-MS-WIN-CRT-HEAP-L1-1-0.DLL ...
----- Loading C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\API-MS-WIN-CRT-HEAP-L1-1-0.DLL -----
```

Figure 3.3: Import Result

We can always call up this information later by going to “Help” > “About `$prog`…” (where `$prog` is the name of the program imported) once we’ve opened the imported file (see below).

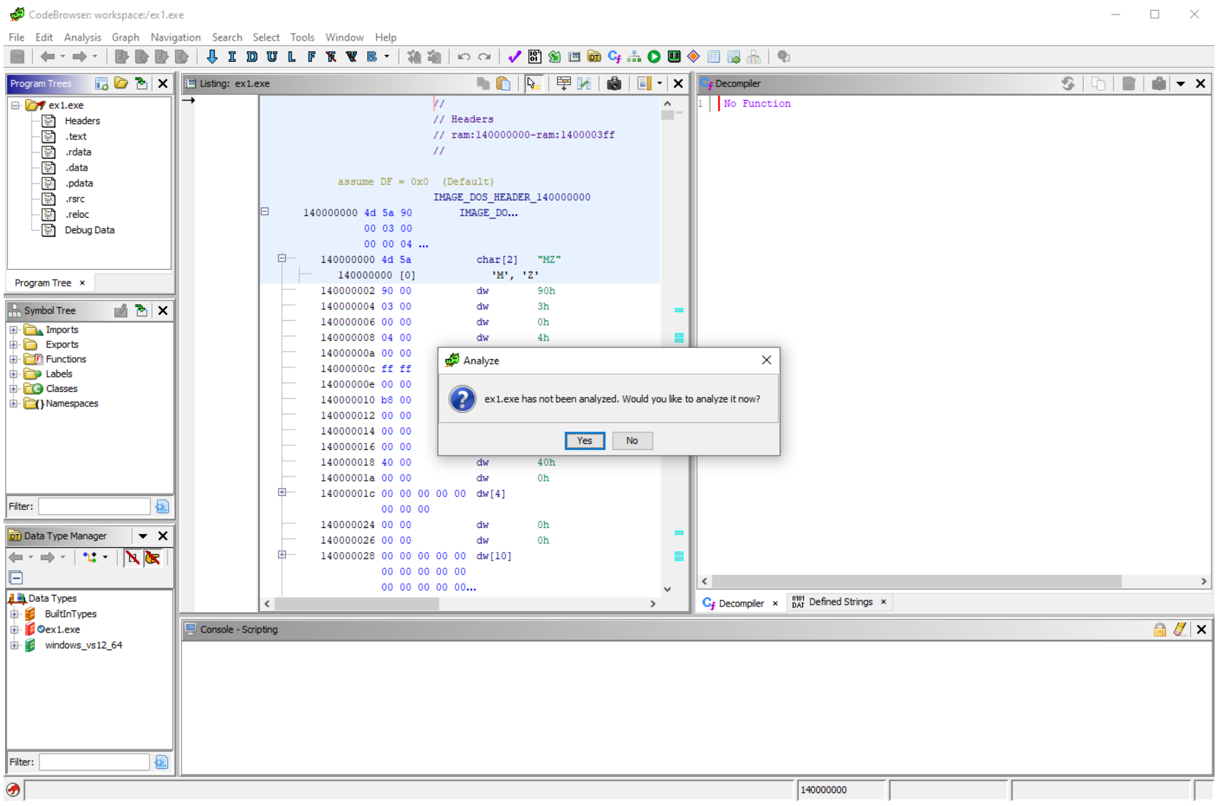
After confirming this as well, we now finally have our program available in *Ghidra*.



Packed database cache: C:\Users\User\AppData\Local\Ghidra\packed-db-cache

Figure 3.4: After Import

Now we can right click on the entry and choose “Open With” > “CodeBrowser” - or simply double click it. After this, a “CodeBrowser” window opens, and we can already see some machine code in the “Listing” component in the background. But first, *Ghidra* would like to analyze the program, which includes disassembling (recreating assembly code from the machine code) and decompiling it (recreating C code). We will confirm and simply accept the default “Analysis Options” in the next dialog to have this analysis take place.



```
                      //
                      // Headers
                      // ram:140000000-ram:1400003ff
                      //

             assume DF = 0x0  (Default)
                      IMAGE_DOS_HEADER_140000000
   140000000 4d 5a 90        IMAGE_DO...
             00 03 00
             00 00 04 ...
     140000000 4d 5a          char[2]   "MZ"
       140000000 [0]            'M', 'Z'
     140000002 90 00          dw        90h
     140000004 03 00          dw        3h
     140000006 00 00          dw        0h
     140000008 04 00          dw        4h
     14000000a 00 00
     14000000c ff ff
     14000000e 00 00
     140000010 b8 00
     140000012 00 00
     140000014 00 00
     140000016 00 00
     140000018 40 00          dw        40h
     14000001a 00 00          dw        0h
     14000001c 00 00 00 00 00 dw[4]
               00 00 00
     140000024 00 00          dw        0h
     140000026 00 00          dw        0h
     140000028 00 00 00 00 00 dw[10]
               00 00 00 00 00
               00 00 00 00 00...
```

```
1  No Function
```

Analyze

ex1.exe has not been analyzed. Would you like to analyze it now?

Figure 3.5: Analysis Dialog

Time to exercise!

3.4.2.2 Settings

I would recommend you make the following changes to the “Code Browser” tool’s settings. To do so, go to “Edit” > “Tool Options” and change the following settings:

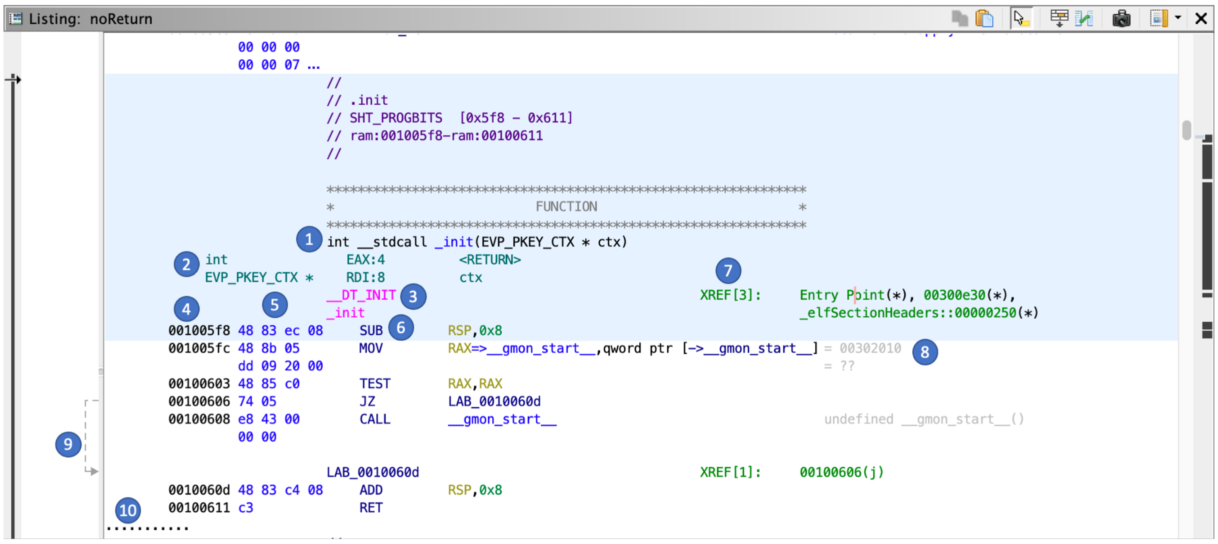
- Auto Analysis: uncheck “Show Analysis Options”
- Decompiler > Analysis: check “Use inplace assignment operators”
- Decompiler > Display: check “Print ‘NULL’ for null pointers”
- Key Bindings > Edit Function Signature: bind to “F”

Once that’s done, close the dialog (“OK”), and persist these settings using “File” > “Save Tool”.

3.4.3 Component Overview

3.4.3.1 Listing

The complete disassembled source of the loaded program is shown in the “Listing” component. Let’s get a feeling for the different information shown.



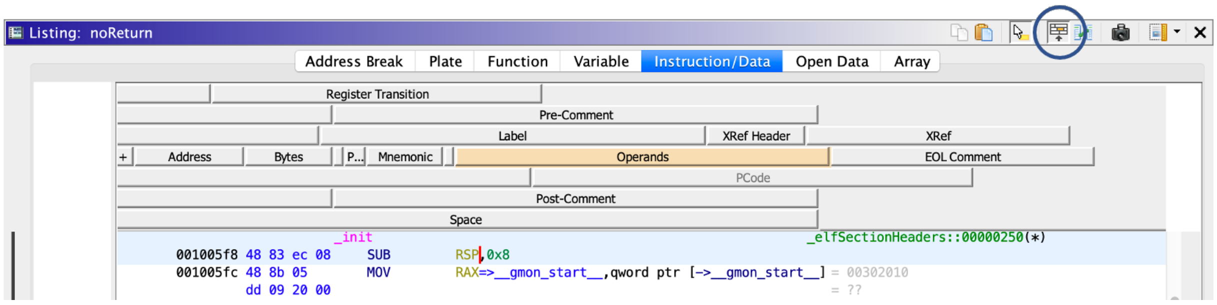
```
                    00 00 00
                    00 00 07 ...
                                //
                                // .init
                                // SHT_PROGBITS  [0x5f8 - 0x611]
                                // ram:001005f8-ram:00100611
                                //

                                **************************************************************
                                *                          FUNCTION                          *
                                **************************************************************
                                int __stdcall _init(EVP_PKEY_CTX * ctx)
              int                 EAX:4         <RETURN>
              EVP_PKEY_CTX *      RDI:8         ctx
                                __DT_INIT                               XREF[3]:     Entry Point(*), 00300e30(*),
                                _init                                                _elfSectionHeaders::00000250(*)
        001005f8 48 83 ec 08        SUB         RSP,0x8
        001005fc 48 8b 05           MOV         RAX=>__gmon_start__,qword ptr [->__gmon_start__] = 00302010
                 dd 09 20 00                                                                      = ??
        00100603 48 85 c0           TEST        RAX,RAX
        00100606 74 05              JZ          LAB_0010060d
        00100608 e8 43 00           CALL        __gmon_start__                               undefined __gmon_start__()
                 00 00

                                LAB_0010060d                            XREF[1]:     00100606(j)
        0010060d 48 83 c4 08        ADD         RSP,0x8
        00100611 c3                 RET
..........
```

Figure 3.6: Listing Component

1. C style function signature
2. Addresses / regististers of parameters, local variables and return value (on stack)
3. Labels
4. Memory address
5. Op codes (machine language)
6. Assembly instructions
7. Incoming references
8. Derived value (end of line comment)
9. Jump / call source -> target
10. Memory “gap” (the dots represent memory locations not shown because they are padding - for alignment to e.g. 8 bytes - only)

The easiest way of getting information about a selected part in the "Listing" component is to call up the field layout editor. It doesn't only allow you to customize the listing layout, but also directly shows which piece of information is currently selected.

As an example, clicking on an operand of an instruction will jump to the "Instruction/Data" tab in the field layout editor and highlight "Operands". If in the screenshot, `SUB` was subsequently clicked, the selection in the editor would jump to "Mnemonic". In order to customize the layout, we could e.g. drag and drop "Bytes" to the left of "Address".

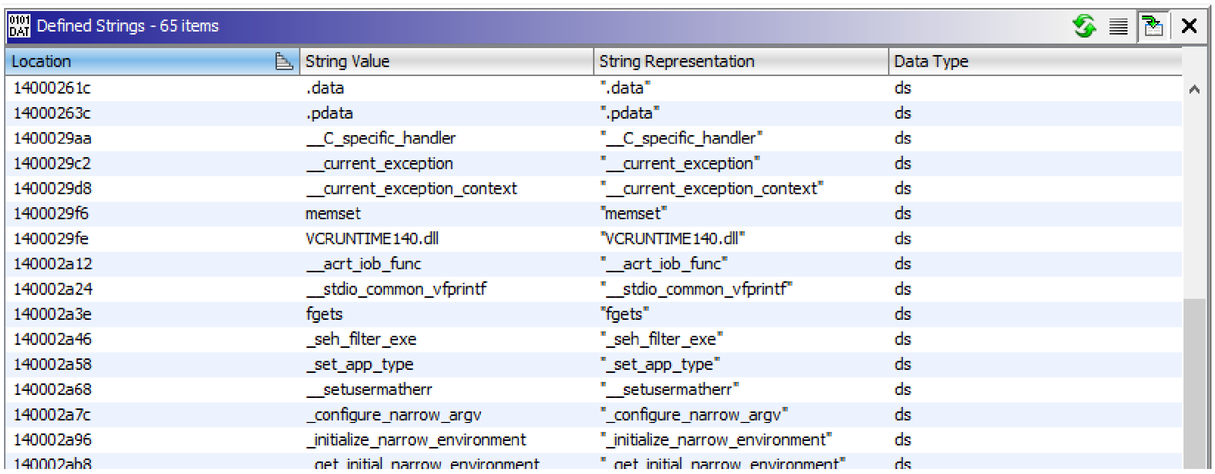


```
                    _init                                         _elfSectionHeaders::00000250(*)
     001005f8 48 83 ec 08     SUB        RSP,0x8
     001005fc 48 8b 05        MOV        RAX=>__gmon_start__,qword ptr [->__gmon_start__] = 00302010
              dd 09 20 00                                                                 = ??
```

Figure 3.7: Fields: Instuction/Data

3.4.3.2 Strings

Ghidra has a component which conveniently extracts a program's strings for us. Open it using "Window" > "Defined Strings".



Location	String Value	String Representation	Data Type
14000261c	.data	".data"	ds
14000263c	.pdata	".pdata"	ds
1400029aa	__C_specific_handler	"__C_specific_handler"	ds
1400029c2	__current_exception	"__current_exception"	ds
1400029d8	__current_exception_context	"__current_exception_context"	ds
1400029f6	memset	"memset"	ds
1400029fe	VCRUNTIME140.dll	"VCRUNTIME140.dll"	ds
140002a12	__acrt_iob_func	"__acrt_iob_func"	ds
140002a24	__stdio_common_vfprintf	"__stdio_common_vfprintf"	ds
140002a3e	fgets	"fgets"	ds
140002a46	_seh_filter_exe	"_seh_filter_exe"	ds
140002a58	_set_app_type	"_set_app_type"	ds
140002a68	__setusermatherr	"__setusermatherr"	ds
140002a7c	_configure_narrow_argv	"_configure_narrow_argv"	ds
140002a96	_initialize_narrow_environment	"_initialize_narrow_environment"	ds
140002ab8	_get_initial_narrow_environment	"_get_initial_narrow_environment"	ds

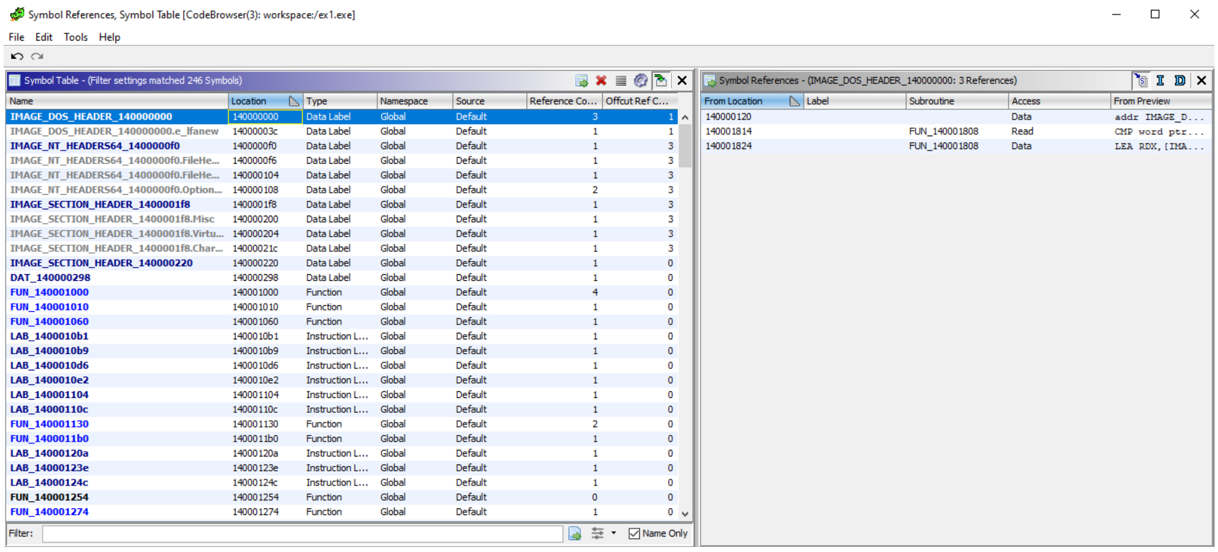
Figure 3.8: Strings Component

This is one of the starting points when reverse engineering programs. If you find something interesting, this might be the fastest way to success. It's unfortunately not (yet) possible to directly navigate to references from the "Defined Strings" component, so we have to double click on the string we're interested in. This will focus the according region in the "Listing" component, from where we can right click and choose "References" > "Show References To Address".

Time to exercise!

3.4.3.3 Symbol References

To generally find references to something we're interested in, we can open the "Symbol References" component, which can list all symbols discovered and show references to them. We may have to change the tool options to see what we're interested in: click on the cogwheel in the "Symbol Table" component and check additional items to include (e.g. "Default (Labels)" to include strings).


Symbol References, Symbol Table [CodeBrowser(3): workspace:/ex1.exe]
File Edit Tools Help

Symbol Table - (Filter settings matched 246 Symbols)

Name	Location	Type	Namespace	Source	Reference Co...	Offcut Ref C...
IMAGE_DOS_HEADER_140000000	140000000	Data Label	Global	Default	3	1
IMAGE_DOS_HEADER_140000000.e_lfanew	14000003c	Data Label	Global	Default	1	1
IMAGE_NT_HEADERS64_1400000f0	1400000f0	Data Label	Global	Default	1	3
IMAGE_NT_HEADERS64_1400000f0.FileHe...	1400000f6	Data Label	Global	Default	1	3
IMAGE_NT_HEADERS64_1400000f0.FileHe...	140000104	Data Label	Global	Default	1	3
IMAGE_NT_HEADERS64_1400000f0.Option...	140000108	Data Label	Global	Default	2	3
IMAGE_SECTION_HEADER_1400001f8	1400001f8	Data Label	Global	Default	1	3
IMAGE_SECTION_HEADER_1400001f8.Misc	140000200	Data Label	Global	Default	1	3
IMAGE_SECTION_HEADER_1400001f8.Virtu...	140000204	Data Label	Global	Default	1	3
IMAGE_SECTION_HEADER_1400001f8.Char...	14000021c	Data Label	Global	Default	1	3
IMAGE_SECTION_HEADER_140000220	140000220	Data Label	Global	Default	1	0
DAT_140000298	140000298	Data Label	Global	Default	1	0
FUN_140001000	140001000	Function	Global	Default	4	0
FUN_140001010	140001010	Function	Global	Default	1	0
FUN_140001060	140001060	Function	Global	Default	1	0
LAB_1400010b1	1400010b1	Instruction L...	Global	Default	1	0
LAB_1400010b9	1400010b9	Instruction L...	Global	Default	1	0
LAB_1400010d6	1400010d6	Instruction L...	Global	Default	1	0
LAB_1400010e2	1400010e2	Instruction L...	Global	Default	1	0
LAB_140001104	140001104	Instruction L...	Global	Default	1	0
LAB_14000110c	14000110c	Instruction L...	Global	Default	1	0
FUN_140001130	140001130	Function	Global	Default	2	0
FUN_1400011b0	1400011b0	Function	Global	Default	1	0
LAB_14000120a	14000120a	Instruction L...	Global	Default	1	0
LAB_14000123e	14000123e	Instruction L...	Global	Default	1	0
LAB_14000124c	14000124c	Instruction L...	Global	Default	1	0
FUN_140001254	140001254	Function	Global	Default	0	0
FUN_140001274	140001274	Function	Global	Default	1	0

Filter: ✓ Name Only

Symbol References - (IMAGE_DOS_HEADER_140000000: 3 References)

From Location	Label	Subroutine	Access	From Preview
140000120			Data	addr IMAGE_D...
140001814		FUN_140001808	Read	CMP word ptr...
140001824		FUN_140001808	Data	LEA RDX,[IMA...

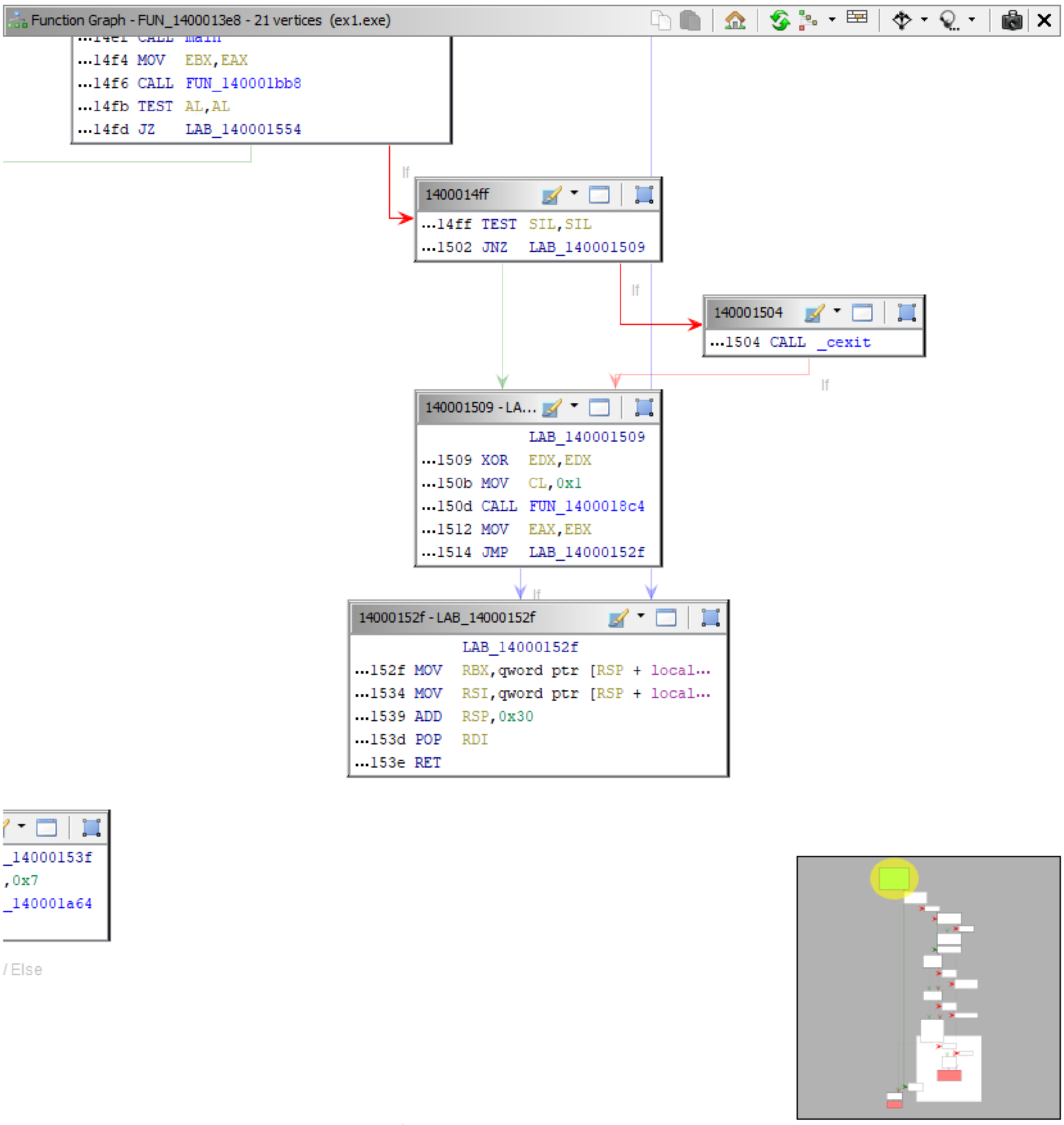
Figure 3.9: Symbol References Component

We can also filter the symbols to e.g. limit our selection to strings discovered and referenced (which seem to get a label starting with "s_").

Time to exercise!

3.4.3.4 Function Graph

The “Function Graph” component gives us a visual representation of the control flow. It groups sequential assembly instructions and uses arrows to visualize the control flow between them.



```
...14f4 MOV   EBX,EAX
...14f6 CALL  FUN_140001bb8
...14fb TEST  AL,AL
...14fd JZ    LAB_140001554
```

```
1400014ff
...14ff TEST  SIL,SIL
...1502 JNZ   LAB_140001509
```

```
140001504
...1504 CALL  _cexit
```

```
140001509 - LA...
              LAB_140001509
...1509 XOR   EDX,EDX
...150b MOV   CL,0x1
...150d CALL  FUN_1400018c4
...1512 MOV   EAX,EBX
...1514 JMP   LAB_14000152f
```

```
14000152f - LAB_14000152f
              LAB_14000152f
...152f MOV   RBX,qword ptr [RSP + local...
...1534 MOV   RSI,qword ptr [RSP + local...
...1539 ADD   RSP,0x30
...153d POP   RDI
...153e RET
```

```
_14000153f
,0x7
_140001a64
```

Figure 3.10: Function Graph Component

3.4.3.5 Symbol Tree

All the way to the left, you will find the component “Symbol Tree”. It lists all the symbols discovered in the program analyzed. The first tree node to focus our attention on is “Exports”. It contains all of the (discovered) symbols accessible from outside of the program. For us, the main use is to have a quick way to the program’s entry point, where execution starts after launch.

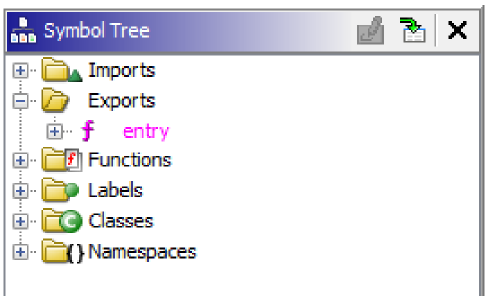


Figure 3.11: Strings Component

If you open the “Functions” node, you will see **all** functions (including the exported ones). Additionally, there is the “Labels” node, which shows all referenced memory locations (including, but not limited to jump targets) that were not identified as functions.

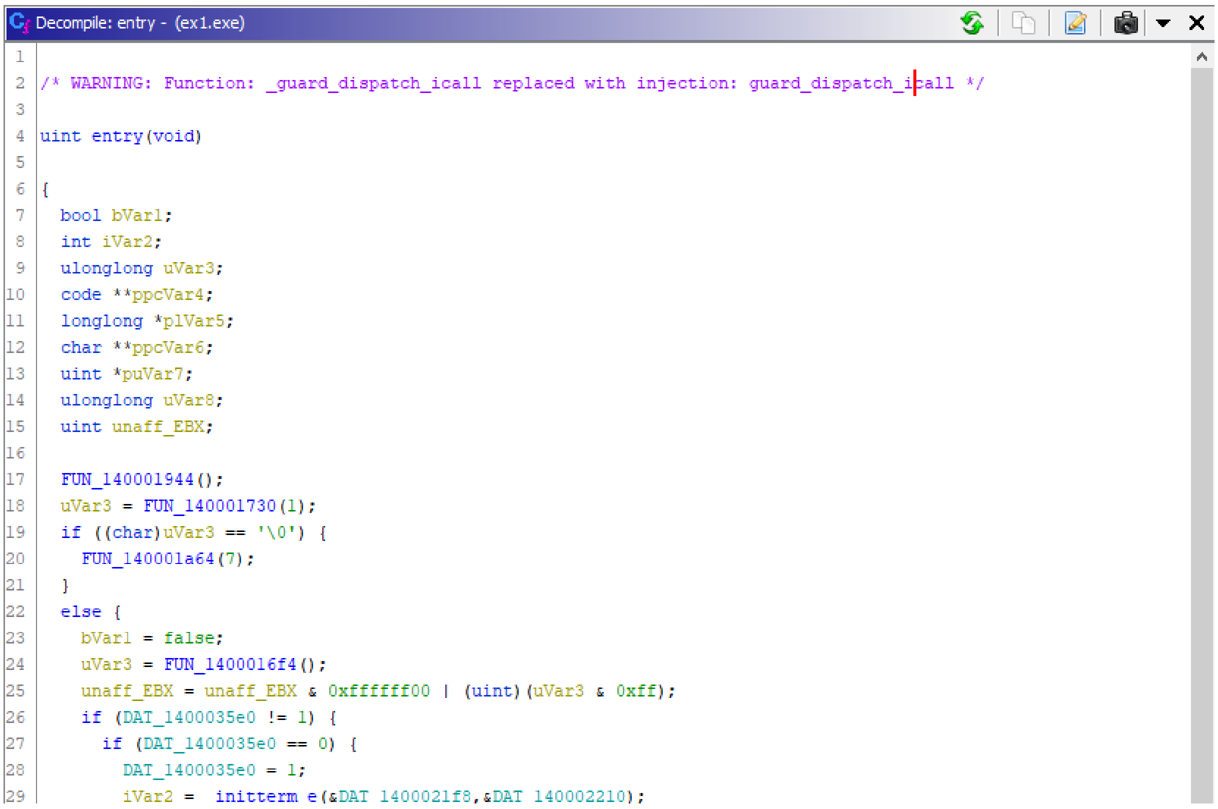
> Functions can be compiled very differently, and it’s not always clear if something was originally a function or just a label created e.g. as the result of a “break” in a loop. Functions are identified using heuristics.

Here are some ways we can move from here:

- By left clicking on one of the tree nodes, *Ghidra* will reveal the according memory position in the “Listing” view.
- Right clicking on a node lets us . . .
  - find all discovered references to the according position (“Show Reference to”).
  - make changes to the function signature (“Edit Function”), e.g. rename it (explained further below).

3.4.3.6 Decompile

The “Decompile” component takes the reverse translation one step further and shows *C* source code of what could have been compiled to the machine code analyzed. This code helps us gain a better understanding of what is happening and is a very good starting point for the analysis of the program we’re looking at.



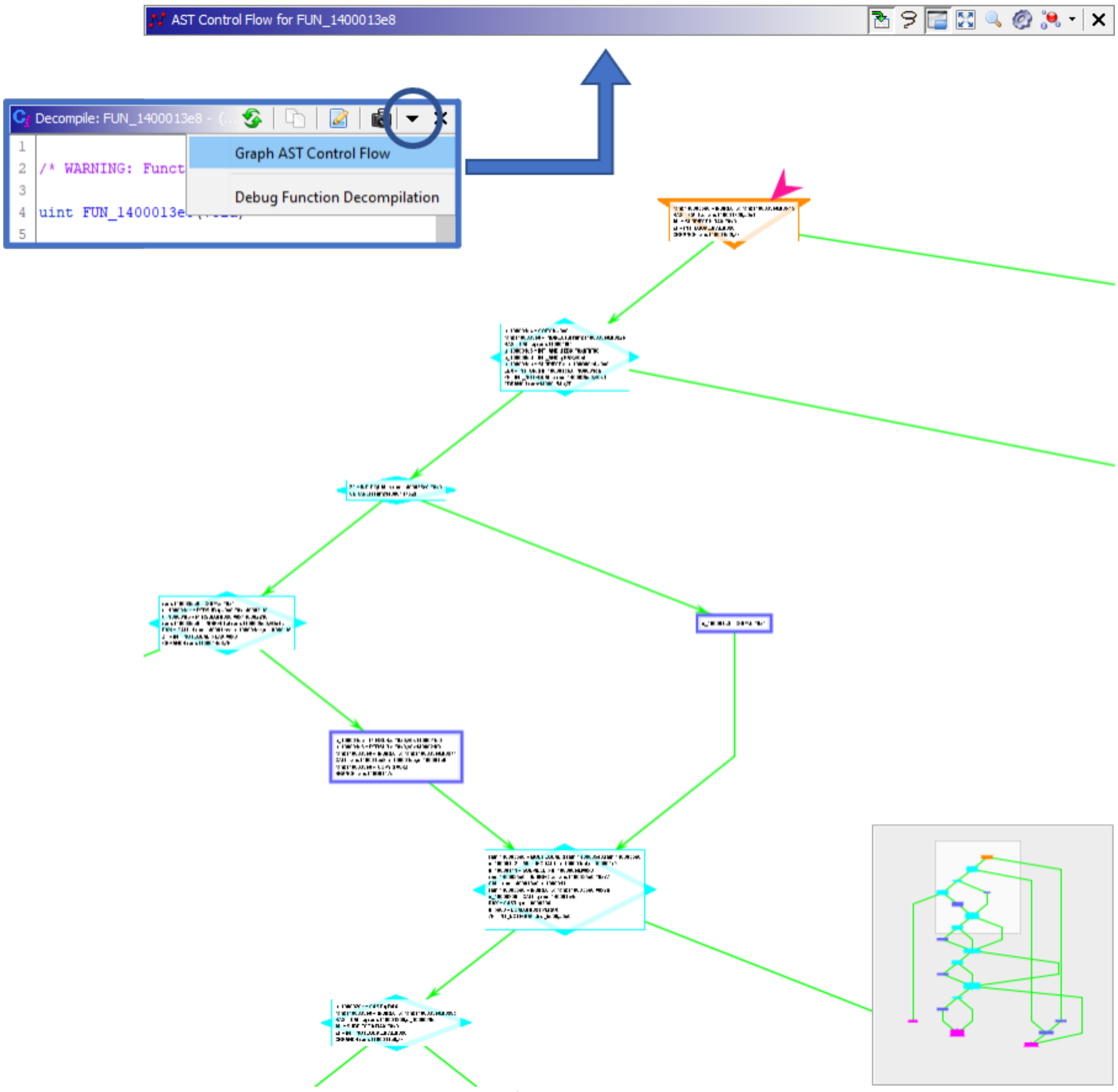
```
/* WARNING: Function: _guard_dispatch_icall replaced with injection: guard_dispatch_icall */

uint entry(void)

{
  bool bVar1;
  int iVar2;
  ulonglong uVar3;
  code **ppcVar4;
  longlong *plVar5;
  char **ppcVar6;
  uint *puVar7;
  ulonglong uVar8;
  uint unaff_EBX;

  FUN_140001944();
  uVar3 = FUN_140001730(1);
  if ((char)uVar3 == '\0') {
    FUN_140001a64(7);
  }
  else {
    bVar1 = false;
    uVar3 = FUN_1400016f4();
    unaff_EBX = unaff_EBX & 0xffffff00 | (uint)(uVar3 & 0xff);
    if (DAT_1400035e0 != 1) {
      if (DAT_1400035e0 == 0) {
        DAT_1400035e0 = 1;
        iVar2 = _initterm_e(&DAT_1400021f8,&DAT_140002210);
```

Figure 3.12: Decompile Component

The code listing shows the last function navigated to in the “Listing” component.

3.4.3.7 AST Control Flow

There is also a visual view of the decompiled code (AST stands for Abstract Syntax Tree [9]), but it's a bit more hidden: in the "Decompiler" component, there's a down arrow which opens a menu. From there, choose "Graph AST Control Flow". this will open an additional component window containing a visual representation of the *C* control flow.


AST Control Flow for FUN_1400013e8
Decompile: FUN_1400013e8
Graph AST Control Flow
Debug Function Decompilation

```
/* WARNING: Funct
uint FUN_1400013e
```

Figure 3.13: AST Control Flow Component

Time to exercise!

[9] https://en.wikipedia.org/wiki/Abstract_syntax_tree

3.4.4 Program Analysis

Once we've imported a program and started the Code browser, we're likely presented with what initially will seem like a bloody mess. And since the best way of dealing with a mess (assuming we can't just ignore it ;-)) is to clean it up, we're going to have a look at how to clean up the code we're presented with using *Ghidra*'s functionality.

> You will often find functions prefixed with "thunk" - you've probably never heard of "thunks", have you? Or, if you have, you might have a different meaning in mind. So, what is a "thunk" in Assembly? It's basically a function that gets `CALL`ed from other locations. What is special about it is that it does some setup (or not) and `JMP`s to another function - instead of `CALL`ing it. This means that when the other function `RET`urns, program execution will go back to the position after where the "thunk" was called!
>
> As an example, if a function `A` `CALL`s a thunk `B`, and `B` then `JMP`s to `C`, when `C` `RET`urns, the program will be back in `A` (and never have passed through `B` on the way back).

3.4.4.1 Navigating & Bookmarks

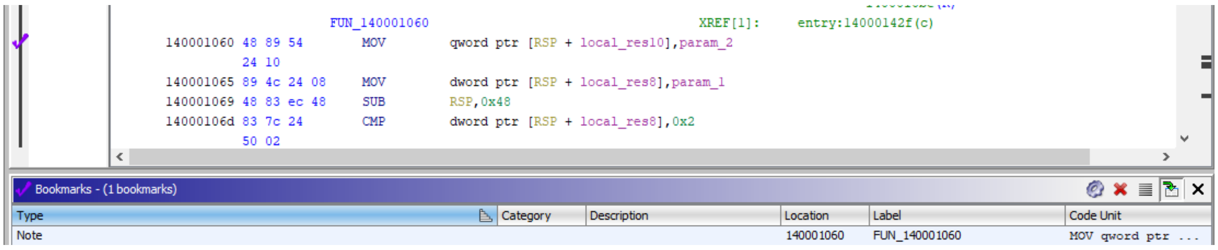
We've already seen that we can navigate to labels by double clicking them in e.g. the "Listing" component. But how do we get back? In the "Navigation" menu entry, there is an item "Previous History Function" you can use to jump back if we navigate into a function. But that's a bit specific, what if we jump to a label which isn't a function?

I would recommend configuring key bindings for "Previous Location in History" (and possibly also "Next Location in History"). To do this, go to "Edit" > "Tool Options" and select "Key Bindings" in the left hand side tree. Search for the entries using the filter and assign a key combination (or remember the one that's already there). I recommend using the combination you also use for the same functionality in the IDE of your choice (if any).

To find our way back to a given place, we can also define *bookmarks* to memory locations: right-click anywhere in the "Listing" component, choose "Bookmark..." and then "OK" in the dialog opened.

> As expected, you can configure keyboard shortcuts for this; the default is "CTRL+D".

If we now open the "Bookmarks" component, we'll have a convenient way back to that location (The type of your bookmark will be "Note").



```
                         FUN_140001060                                    XREF[1]:     entry:14000142f(c)
     140001060 48 89 54        MOV        qword ptr [RSP + local_res10],param_2
               24 10
     140001065 89 4c 24 08     MOV        dword ptr [RSP + local_res8],param_1
     140001069 48 83 ec 48     SUB        RSP,0x48
     14000106d 83 7c 24        CMP        dword ptr [RSP + local_res8],0x2
               50 02
```

Bookmarks - (1 bookmarks)

Type	Category	Description	Location	Label	Code Unit
Note			140001060	FUN_140001060	MOV qword ptr ...

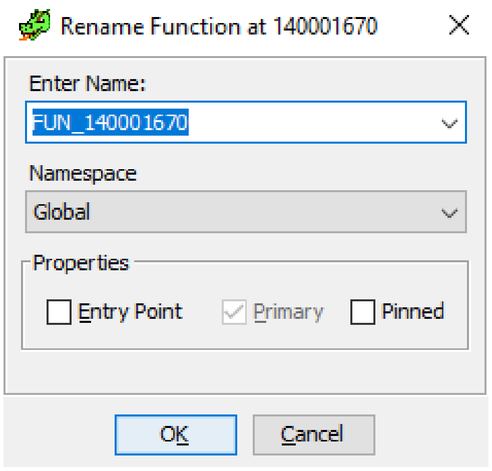
Figure 3.14: Bookmarks Component

3.4.4.2 Renaming Functions

Since we typically don't get debug symbols along with the programs we analyze, *Ghidra* will only be able to show us rather unhelpful names for the functions we encounter. Once we figure out what a function does, it would be nice if we could reflect this directly in *Ghidra* - good thing we can. How to do it depends on where you start from:

- In the "Listing" component, you can right-click on the ...
  - function signature (the *C* style one) and choose "Function" > "Rename Function"
  - function's label and choose "Edit Label"
- In the "Decompile" component, right-click on the function name and choose "Rename Function"

Generally, you can just type a lower case "L" to get the same dialog.


Rename Function at 140001670

Enter Name: FUN_140001670

Namespace: Global

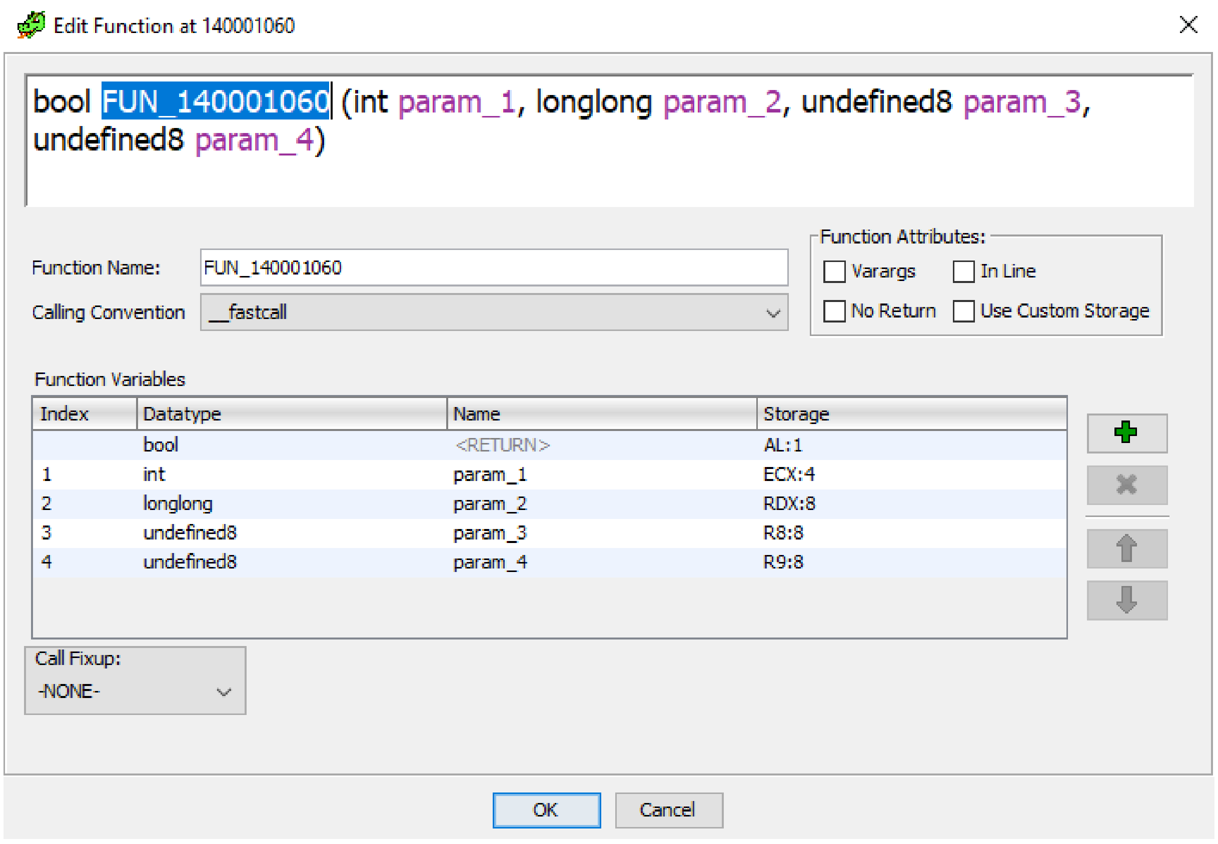
Properties: [ ] Entry Point [✓ Primary] [ ] Pinned

Figure 3.15: Rename Function Action

3.4.4.3 Editing Function Signatures

We will often also want to correct the complete derived function signature instead of just the function's name. For example, we might want to have our `main` function read as `int main(int argc, char** argv)` (in *Ghidra*, you add type (`char**`) and name (`argv`) into two separate fields, so the syntax `char* argv[]` doesn't work here) instead of e.g. `bool FUN_140001060(int param_1,longlong param_2)` or `undefined entry()`. We can do this by:

- In the "Listing" component, right-clicking on the function signature and choosing "Edit Function" (or typing lower case "F")
- In the "Decompile" component, right-clicking on the function name and choosing "Edit Function Signature" (I would recommend adding a key binding to "F" in "Edit" > "Tool Options", "Key Bindings")


Edit Function at 140001060

```
bool FUN_140001060(int param_1, longlong param_2, undefined8 param_3,
undefined8 param_4)
```

Function Name: FUN_140001060
Calling Convention: __fastcall

Function Attributes: Varargs [ ] In Line [ ] No Return [ ] Use Custom Storage [ ]

Function Variables

Index	Datatype	Name	Storage
	bool	<RETURN>	AL:1
1	int	param_1	ECX:4
2	longlong	param_2	RDX:8
3	undefined8	param_3	R8:8
4	undefined8	param_4	R9:8

Call Fixup: -NONE-

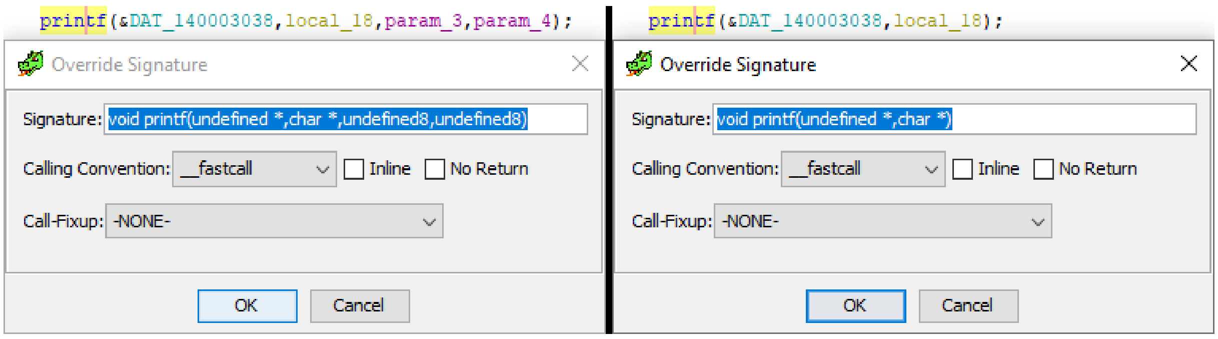
Figure 3.16: Edit Function Action

The entire signature can be edited directly in the topmost text field, or individual parameters can be edited one by one.

Time to exercise!

3.4.4.4 Overriding Call Signatures

Once we've corrected a function's signature, all of the callers should be correct as well, right? Mostly, they will be. But if a function accepts a variable number of arguments (*Ghidra* expresses this by assigning a function a *varargs* attribute as you may already have seen in the exercises), we may still have function calls around which have been identified to pass an incorrect amount of parameters. We can override the call to change which arguments are passed. Note that this will only work in the "Decompile" component - the action does not exist in the "Listing" component.



```
printf(&DAT_140003038,local_18,param_3,param_4);
```

Override Signature

Signature: void printf(undefined *,char *,undefined8,undefined8)

Calling Convention: __fastcall ☐ Inline ☐ No Return

Call-Fixup: -NONE-

```
printf(&DAT_140003038,local_18);
```

Override Signature

Signature: void printf(undefined *,char *)

Calling Convention: __fastcall ☐ Inline ☐ No Return

Call-Fixup: -NONE-

Figure 3.17: Override Function Call Action (before & after)

Time to exercise!

3.4.4.5 Renaming & Retyping Variables

Similarly to functions, we can rename variables in the "Decompile" component to help us better understand the code we're working on. To do so, we right-click on a variable and select "Rename Variable" (or type a lower case "L"). Similarly, to redefine the type of a variable, we can choose "Retype Variable" (CTRL + L / CMD + L).

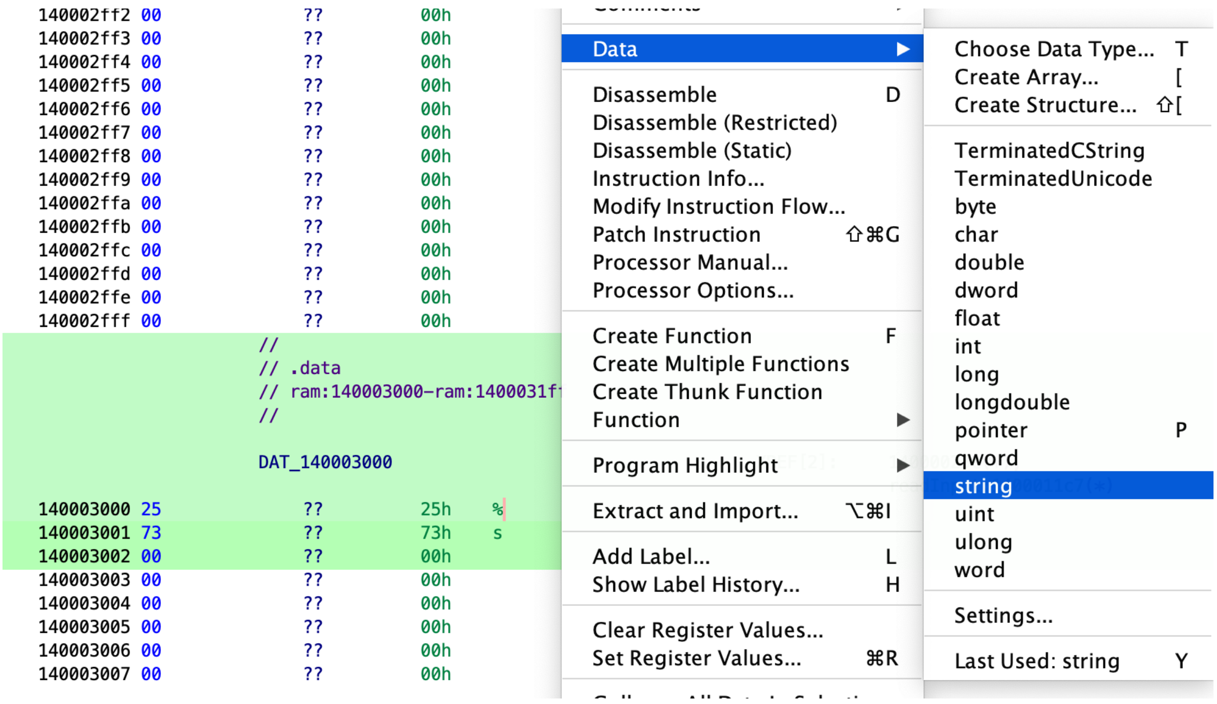
Unfortunately, renaming variables currently has an unexpected side effect: *Ghidra* replaces register names by the assigned variable name in the "Listing" component as well - and doesn't reset them even when the register is used for something else. See issue #2827 [10] for details.

Time to exercise!

[10] https://github.com/NationalSecurityAgency/ghidra/issues/2827

3.4.4.6 Retyping Data

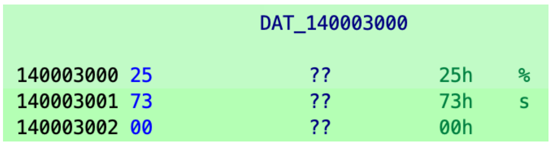
When we find data which hasn't been recognized for what it is, we can also change that by selecting the region in question, right clicking on it and choosing the appropriate type from "Data".



```
140002ff2 00          ??       00h
140002ff3 00          ??       00h
140002ff4 00          ??       00h
140002ff5 00          ??       00h
140002ff6 00          ??       00h
140002ff7 00          ??       00h
140002ff8 00          ??       00h
140002ff9 00          ??       00h
140002ffa 00          ??       00h
140002ffb 00          ??       00h
140002ffc 00          ??       00h
140002ffd 00          ??       00h
140002ffe 00          ??       00h
140002fff 00          ??       00h
                  //
                  // .data
                  // ram:140003000-ram:1400031f
                  //

                  DAT_140003000

140003000 25          ??       25h     %
140003001 73          ??       73h     s
140003002 00          ??       00h
140003003 00          ??       00h
140003004 00          ??       00h
140003005 00          ??       00h
140003006 00          ??       00h
140003007 00          ??       00h
```

Figure 3.18: Edit Data Type Menu

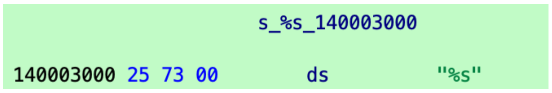
As an example, the following format string was unrecognized:



```
                      DAT_140003000

140003000 25              ??        25h     %
140003001 73              ??        73h     s
140003002 00              ??        00h
```

Figure 3.19: Data with Wrong Type

When we choose "string" as a new data type, we get the following result:



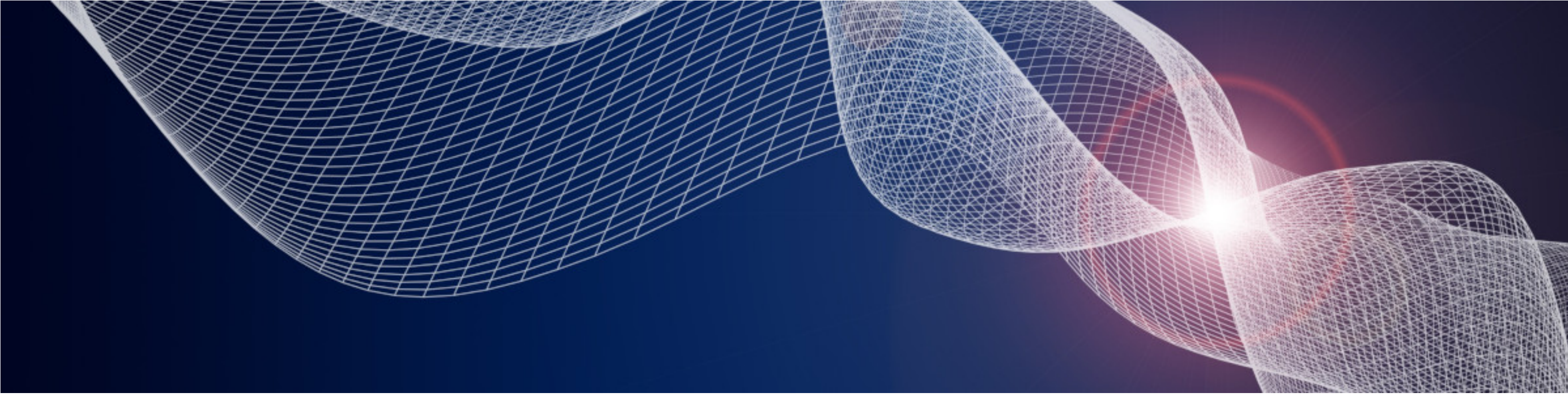
```
                      s_%s_140003000

140003000 25 73 00        ds           "%s"
```

Figure 3.20: Data with Corrected Type

3.4.4.7 Renaming Labels

We can also rename the new label generated when we corrected the data type above by right-clicking and choosing "Edit Label" (or pressing lower case "L").

Time to exercise!



4. Dynamic Analysis

4.1 Introduction

So far, we've tried to understand what code does without ever running it. And since humans are of course much worse in interpreting code than machines are, lots of things take much longer than if we could just run a program to see what it does.

This is where dynamic analysis comes into play. In the same way as we debug code we write ourselves, we can debug programs which have been written by others and for which we don't have the sources. The only problem with this is... that we don't have the sources. So we will again have to rely on disassembled and decompiled code.

Fortunately, we can stay within the environment we've come to know: *Ghidra* allows us to connect to debuggers (which in fact are programs that will control which instructions to run and are able to inspect the memory used by the application being executed) and to work with the same views we're used to.

We will use *Windows* executables as examples (most malware is still aimed at this OS) with *WinDbg* as connected debugger, but what we do should equally work in *UNIX* using *GDB* in the background.

4.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Basics	15	0	15
Component Overview	45	90	135
Program Analysis	45	240	325
Patching Programs	15	30	45
Total	**120**	**360**	**480**

4.3 Basics

4.3.1 Ghidra Tools

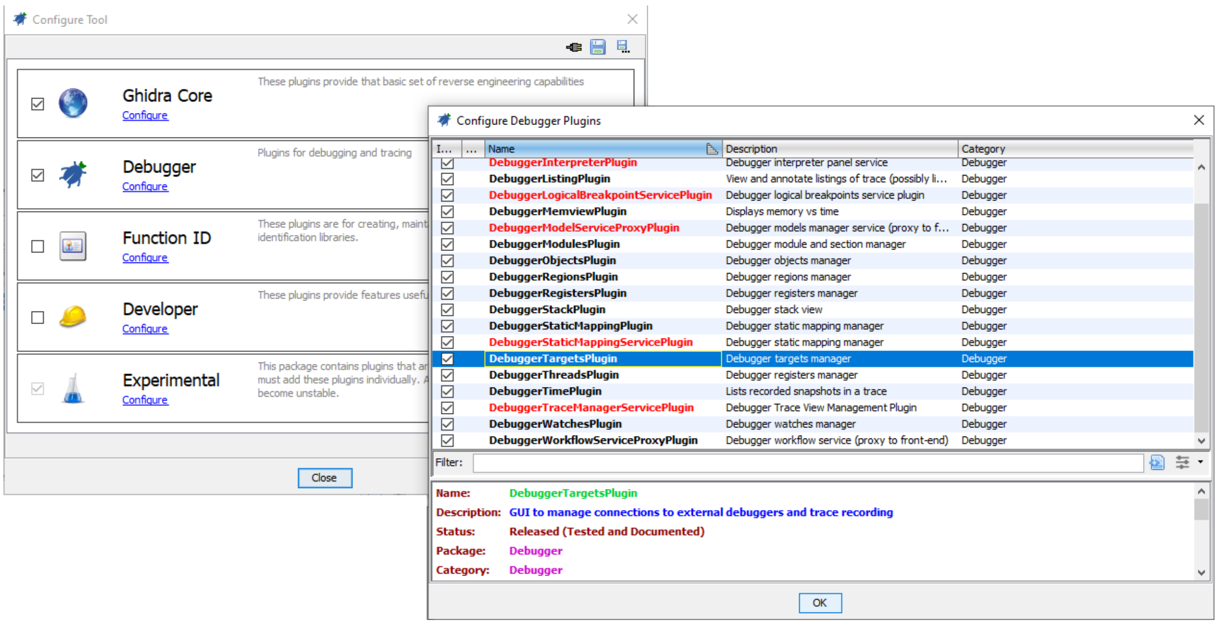
In the "Static Analysis" chapter, we have only used the "Code Browser" tool, and we didn't really think about what a tool is. Now that we're also looking at another tool, the "Debugger", we should understand what these are.

In a nutshell, a *Ghidra* "Tool" is a collection of components, their settings and a layout defining where they are shown and how much space they take up. We've already seen how available components are added to a tool ("Window" > ("Debugger" >) "$component") and how they are configured ("Edit" > "Tool Options"). We can further customize a tool by dragging components around (clicking and holding by their title bar to "pick them up"):

- we can create a group of components using the same area by dragging one component onto another (aim for the center of the target component)
- we can split the area occupied by one component (`A`) and share it with an additional one (`B`) by dragging `B` to one of the edges of `A`.
- we can create a separate, floating window by dragging the component anywhere where there is no other component.

4.3.2 Enabling Components

To use a component which is not part of the default defined for a tool, we can still enable it. To do so, go to "File" > "Configure" and click on the "configure" link in the appropriate plugin section. Components can be enabled by clicking the appropriate checkboxes.


Configure Tool

☑	Ghidra Core	Configure	These plugins provide that basic set of reverse engineering capabilities
☑	Debugger	Configure	Plugins for debugging and tracing
☐	Function ID	Configure	These plugins are for creating, maint identification libraries.
☐	Developer	Configure	These plugins provide features usefu
☑	Experimental	Configure	This package contains plugins that ar must add these plugins individually. A become unstable.

Close

Configure Debugger Plugins

I...	...	Name	Description	Category
☑		DebuggerInterpreterPlugin	Debugger interpreter panel service	Debugger
☑		DebuggerListingPlugin	View and annotate listings of trace (possibly li...	Debugger
☑		DebuggerLogicalBreakpointServicePlugin	Debugger logical breakpoints service plugin	Debugger
☑		DebuggerMemviewPlugin	Displays memory vs time	Debugger
☑		DebuggerModelServiceProxyPlugin	Debugger models manager service (proxy to f...	Debugger
☑		DebuggerModulesPlugin	Debugger module and section manager	Debugger
☑		DebuggerObjectsPlugin	Debugger objects manager	Debugger
☑		DebuggerRegionsPlugin	Debugger regions manager	Debugger
☑		DebuggerRegistersPlugin	Debugger registers manager	Debugger
☑		DebuggerStackPlugin	Debugger stack view	Debugger
☑		DebuggerStaticMappingPlugin	Debugger static mapping manager	Debugger
☑		DebuggerStaticMappingServicePlugin	Debugger static mapping manager	Debugger
☑		DebuggerTargetsPlugin	Debugger targets manager	Debugger
☑		DebuggerThreadsPlugin	Debugger registers manager	Debugger
☑		DebuggerTimePlugin	Lists recorded snapshots in a trace	Debugger
☑		DebuggerTraceManagerServicePlugin	Debugger Trace View Management Plugin	Debugger
☑		DebuggerWatchesPlugin	Debugger watches manager	Debugger
☑		DebuggerWorkflowServiceProxyPlugin	Debugger workflow service (proxy to front-end)	Debugger

Filter:

Name: DebuggerTargetsPlugin
Description: GUI to manage connections to external debuggers and trace recording
Status: Released (Tested and Documented)
Package: Debugger
Category: Debugger

OK

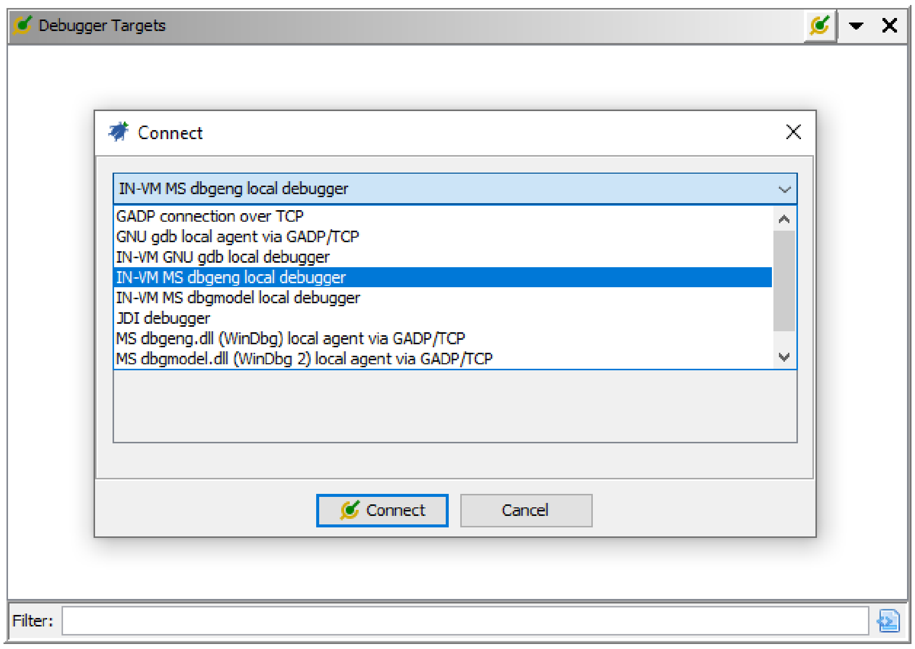
Figure 4.1: Tool Configuration

4.4 Component Overview

As we go along, looking at each component, please enable the different components in *Ghidra*, start a program in the debugger and make sure you see the expected information in the components we look at.

4.4.1 Debugger Targets

The component “Debugger Targets” is where *Ghidra* connects to different debuggers. We use this to start a debugger session for the program we’re interested in. To do so, we click on the “Connect” button, which opens a dialog in which we can select the appropriate way to start / connect to a debugger.



```
GADP connection over TCP
GNU gdb local agent via GADP/TCP
IN-VM GNU gdb local debugger
IN-VM MS dbgeng local debugger
IN-VM MS dbgmodel local debugger
JDI debugger
MS dbgeng.dll (WinDbg) local agent via GADP/TCP
MS dbgmodel.dll (WinDbg 2) local agent via GADP/TCP
```

Figure 4.2: Debugger Targets Component

Depending on which OS we’re working on, we will typically choose to use either *GDB* (on *Unix*) or WinDbg (on *Windows*, represented in *Ghidra* as “dbgeng”, the engine underlying *WinDbg*). There are two ways we can launch and connect to these debuggers:

1. Connect directly from within the same ***J**ava **V**irtual **M**achine* (JVM) in which *Ghidra* is running - choose the “IN-VM [. . .]” option for this.
2. Create a separate process running in a separate JVM - choose the “[. . .] local agent via GADP/TCP” option for this.

In theory, both alternatives should lead to the same result, but in practice, there were differences at the time of writing. With the “IN-VM” option, there is a risk that if something goes wrong in a debugger session running in the same VM as *Ghidra*, *Ghidra* will die along with it, so the “agent” option is preferable if it works. If it doesn’t, use the “IN-VM” option.

Make sure in "File" > "Configure" > "Debugger" you’ve enabled "DebuggerTargetsPlugin"!

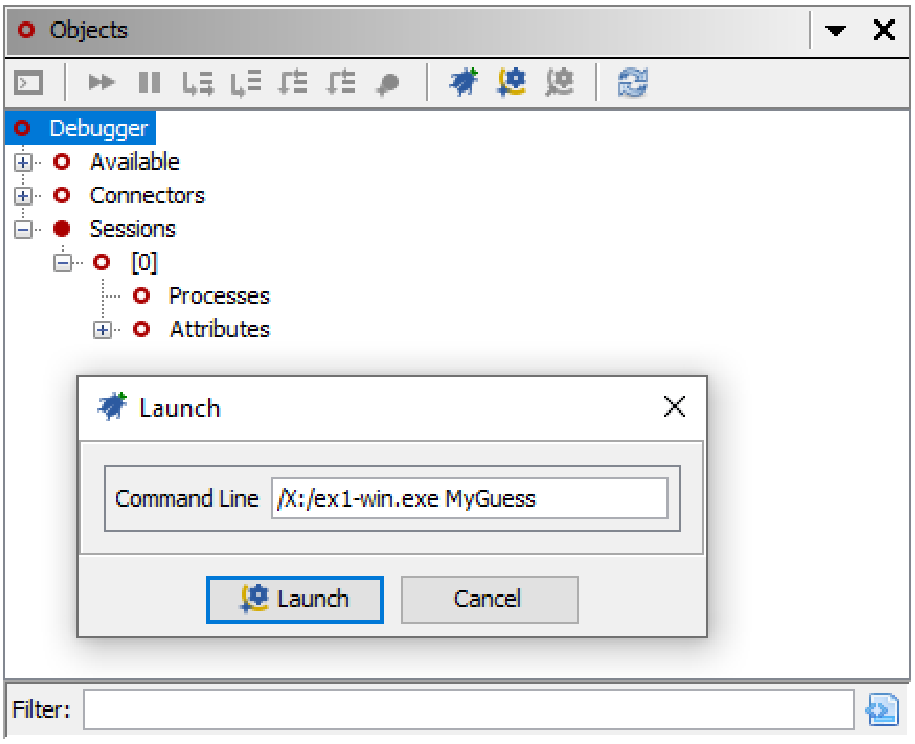
4.4.2 Objects

The “Objects” component is where we load the programs we want to debug into the connected debugger. It shows us the different debugger sessions and lists the processes / threads of each connected program.

To launch a program to be debugged, right-click the “Debugger” node and select “Exec” (or type lower case “X”). In the dialog that pops up, enter the command to launch your program; this allows providing command line arguments. If you don’t remember where the program was imported from, you can go to “Help” > “About $prog” and copy the path from “Executable Location”. You can additionally provide command line arguments to the program - the same way you would call the program from a console.

Once launched, the program is paused immediately, giving us the chance to do any preparatory work such as installing breakpoints (see below).

In *Windows*, both forward and backward slashes are allowed and the path can be prefixed by a forward slash (as shown in “Help” > “About /$prog”).



Command Line /X:/ex1-win.exe MyGuess

Figure 4.3: Objects Component

Make sure in "File" > "Configure" > "Debugger" you’ve enabled "DebuggerObjectsPlugin"!

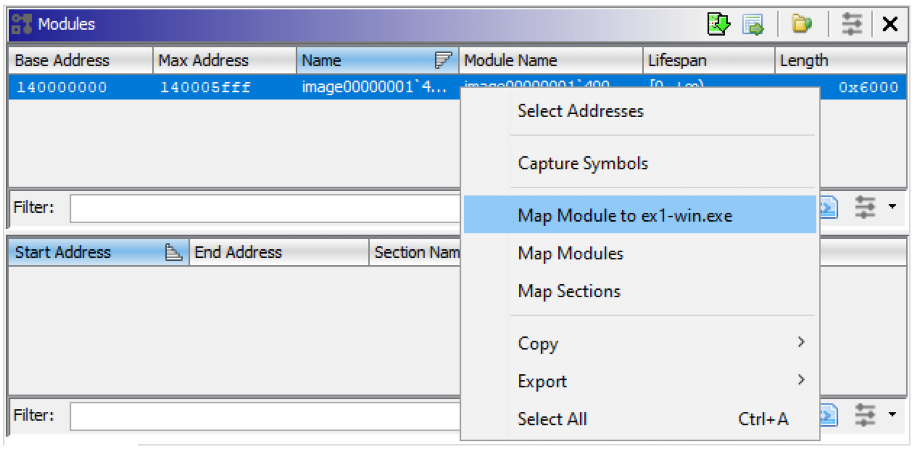
4.4.3 Interpreter

When a new debugger session is started, an “Interpreter” window is typically opened. If it doesn’t - or if we close it later - we can reopen it from the top left icon in the “Objects” component (it is enabled along with that). This console gives us a possibility to directly work with the native debugger connected. This lets us use features which are not (yet) mapped in *Ghidra* - the downside is that we need to handle different debuggers differently.

4.4.4 Modules

The “Modules” component allows virtual memory addresses of the program loaded into memory to be mapped to the program’s and its used libraries’ memory ranges. Only once this is done will the debugger be able to relate addresses in e.g. a listing component (where, as we’ll see, we e.g. can set breakpoints) to addresses in the running program.

At the beginning of an execution, when a program has just been loaded, there will typically only be one “module” shown. Others will be added as the program executes and external libraries are loaded. Right click on this initially visible line and choose “Map Module to `$prog`”, where `$prog` is the name of the current program.



Base Address	Max Address	Name	Module Name	Lifespan	Length
140000000	140005fff	image00000001`4...	image00000001`400	[0, +oo)	0x6000

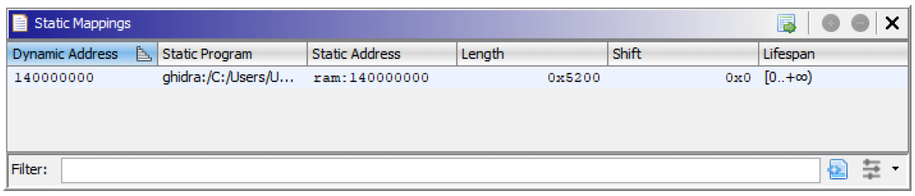
Figure 4.4: Modules Component

Should there not be any entry shown in the “Modules” component for a running program connected to by a debugger session, restart *Ghidra*.

Make sure in "File" > "Configure" > "Debugger" you’ve enabled "DebuggerModulesPlugin"!

4.4.5 Static Mappings

Once we've mapped our program's addresses, a mapping should show up in the "Static Mappings" component. It's useful to check this entry exists before going on - breakpoints (see below) will not work if this is not properly set up.



Dynamic Address	Static Program	Static Address	Length	Shift	Lifespan
140000000	ghidra:/C:/Users/U...	ram:140000000	0x5200	0x0	[0..+∞)

Figure 4.5: Static Mappings Component

Make sure in "File" > "Configure" > "Debugger" you've enabled "DebuggerStaticMappingPlugin"!

4.4.6 Breakpoints

Breakpoints allow us to interrupt program execution when something happens. There are two different ways breakpoints can be implemented:

- Software breakpoints work by changing the machine code being executed and injecting very specific instructions which will interrupt execution of the program being debugged and transfer control to the debugger.
- Hardware breakpoints work by making use of dedicated CPU registers (on architectures where they're available - which is typically the case these days). The CPU will compare these registers' contents to the *instruction pointer* or the address being read from or written to by an instruction and interrupt execution when they match.

Breakpoints can be set from the "Listing" component, where they will also be indicated by both a breakpoint marker and highlighting the line on which they are set. *Ghidra* allows us to set these different types of breakpoints:

- Execution, which will interrupt execution when the marked instruction is executed
  - SOFTWARE (software breakpoint)
  - EXECUTE (hardware breakpoint)
- Memory access, which will interrupt execution when the marked memory location is read from or written to
  - READ (hardware breakpoint)
  - WRITE (hardware breakpoint)
  - READ/WRITE (hardware breakpoint)

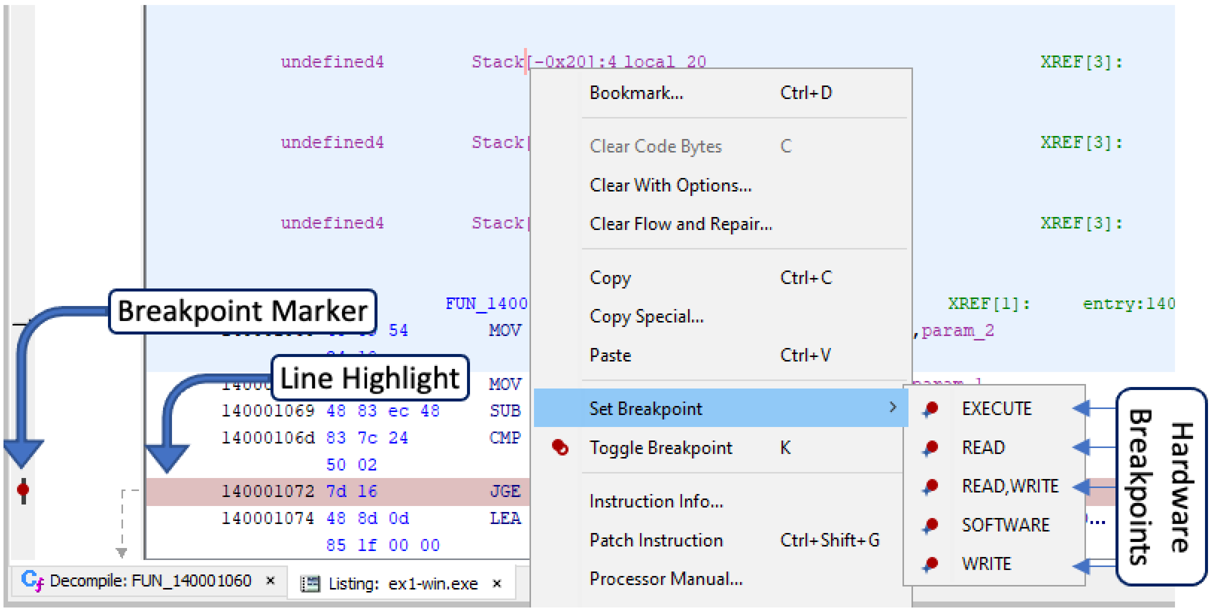
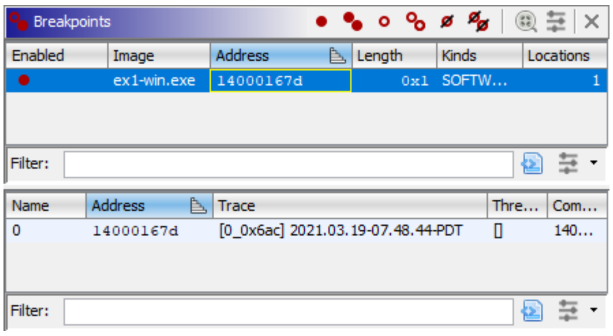


Figure 4.6: Set Breakpoint Action

Defined breakpoints will also be shown in the “Breakpoints” component. In the top half of the component, the breakpoints we set will be shown. The lower half of the component allows us to verify that these defined breakpoints have been successfully mapped to code. If there’s no entry there, this likely means that we

- have not yet launched the program we’re looking at.
- haven’t mapped the appropriate module or did the mapping after the breakpoint was already defined.
- the breakpoints are disabled.

In the last two cases, simply enable the breakpoint(s) by clicking on the button with the filled-in red circle (to enable a single, selected breakpoint) or the one with two of these (to enable all breakpoints).



Enabled	Image	Address	Length	Kinds	Locations
●	ex1-win.exe	14000167d	0x1	SOFTW...	1

Name	Address	Trace	Thre...	Com...
0	14000167d	[0_0x6ac] 2021.03.19-07.48.44-PDT	[]	140...

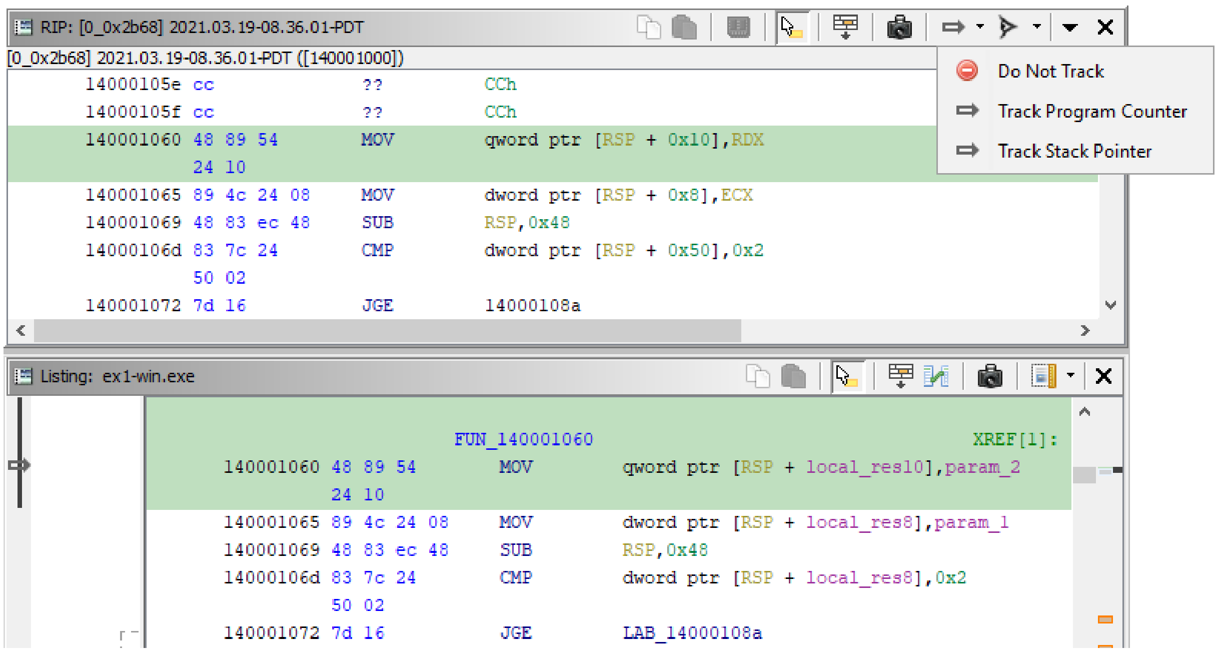
Figure 4.7: Breakpoints Component

Make sure in "File" > "Configure" > "Debugger" you’ve enabled "DebuggerBreakpointsPlugin" & "DebuggerBreakpointMarkerPlugin"!

4.4.7 Dynamic Listing

The “Dynamic Listing” component is the debugger’s equivalent of the “Listing” component we already saw in the “Static Analysis” chapter. There are two main differences:

- The “Dynamic Listing” component can be configured to always navigate to the location the control flow is currently interrupted at.
- The disassembled instructions are shown “as is”, so there is e.g. no variable renaming done. As an example, see below for a comparison between the two components.



```
[0_0x2b68] 2021.03.19-08.36.01-PDT ([140001000])
      14000105e cc              ??        CCh
      14000105f cc              ??        CCh
      140001060 48 89 54        MOV       qword ptr [RSP + 0x10],RDX
                24 10
      140001065 89 4c 24 08     MOV       dword ptr [RSP + 0x8],ECX
      140001069 48 83 ec 48     SUB       RSP,0x48
      14000106d 83 7c 24        CMP       dword ptr [RSP + 0x50],0x2
                50 02
      140001072 7d 16           JGE       14000108a
```

```
                          FUN_140001060                          XREF[1]:
      140001060 48 89 54        MOV       qword ptr [RSP + local_res10],param_2
                24 10
      140001065 89 4c 24 08     MOV       dword ptr [RSP + local_res8],param_1
      140001069 48 83 ec 48     SUB       RSP,0x48
      14000106d 83 7c 24        CMP       dword ptr [RSP + local_res8],0x2
                50 02
      140001072 7d 16           JGE       LAB_14000108a
```

Figure 4.8: Dynamic Listing vs. (Static) Listing Component

Using the “Track Location” button, we can decide whether or not the “Dynamic Listing” component will track

- the *instruction pointer* (called “Program Counter” in *Ghidra*), following the program’s flow.
- the *stack pointer*, following the current stack frame.
- nothing, in a way behaving like a static “Listing” component without register / location renaming.

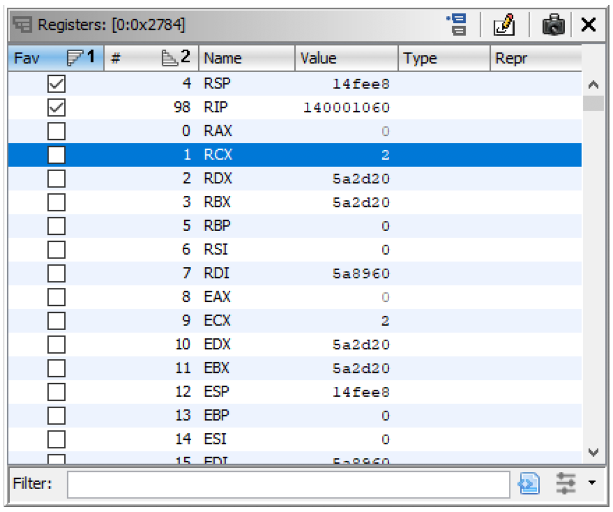
Very often we’re interested in tracking both program execution and stack memory, so it’s nice that we can open two instances of the “Dynamic Listing” component at the same time, setting one each to track the two at once.

> The dynamic listing doesn’t necessarily show disassembled instructions, it may just display `??` next to the bytes. To change this, click on the `??` of the current instruction and press lower case “D”. This disassembles instructions starting from the selected one.

Make sure in "File" > "Configure" > "Debugger" you’ve enabled "DebuggerListingPlugin"! Do NOT switch it on while debugging - it may fail otherwise.

4.4.8 Registers

While code is being executed, we can look at the values which are currently stored in the CPU's registers in the "Registers" component.


Registers: [0:0x2784]

Fav	#	Name	Value	Type	Repr
✓	4	RSP	14fee8		
✓	98	RIP	140001060		
☐	0	RAX	0		
☐	1	RCX	2		
☐	2	RDX	5a2d20		
☐	3	RBX	5a2d20		
☐	5	RBP	0		
☐	6	RSI	0		
☐	7	RDI	5a8960		
☐	8	EAX	0		
☐	9	ECX	2		
☐	10	EDX	5a2d20		
☐	11	EBX	5a2d20		
☐	12	ESP	14fee8		
☐	13	EBP	0		
☐	14	ESI	0		
☐	15	EDI	5a8960		

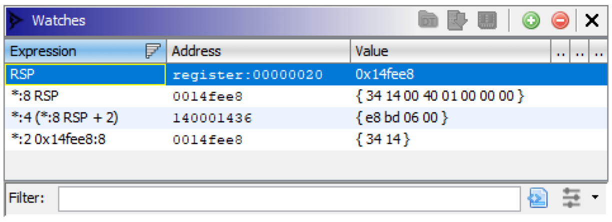
Filter:

Figure 4.9: Registers Component

Make sure in "File" > "Configure" > "Debugger" you've enabled "DebuggerRegistersPlugin"!

4.4.9 Watches

The "Watches" component lets us evaluate register and memory contents. To access a value, we can add a new "watch" and edit its expression to refer to the value we're interested in.


Watches

Expression	Address	Value
RSP	register:00000020	0x14fee8
*:8 RSP	0014fee8	{ 34 14 00 40 01 00 00 00 }
:4 (:8 RSP + 2)	140001436	{ e8 bd 06 00 }
*:2 0x14fee8:8	0014fee8	{ 34 14 }

Filter:

Figure 4.10: Watches Component

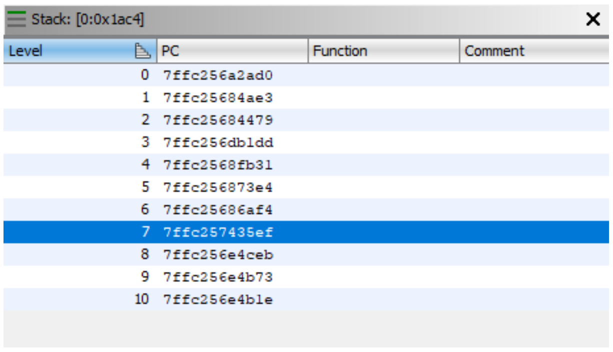
In the example, we see four different expressions:

- `RSP`: evaluates the content of register `RSP`.
- `*:8 RSP`: evaluates 8 bytes of memory content at location `RSP`.
- `*:4 (*:8 RSP + 2)`: evaluates 4 bytes of memory content at the location referred to by the 8 bytes of memory content at location `RSP + 2`.
- `*:2 0x14fee8:8`: evaluates 2 bytes of memory content at location `0x14fee8`.
  - The `:8` is required even though redundant - it's a known issue according to the documentation.

Make sure in "File" > "Configure" > "Debugger" you've enabled "DebuggerWatchesPlugin"!

4.4.10 Stack

The “Stack” component gives you an overview of the stack frames created by the functions in the current thread. As you know, each `CALL` instruction will create a new stack frame, which will be cleared by a later `RET` instruction. Each line represents one stack frame. The column “Level” orders the stack frames, with `0` for the current one, “PC” shows the starting address of function `CALL`ed, “Function” will be filled in if a label is associated with that address, and “Comment” allows you to add a note to that stack frame (double click the cell and write).


Stack: [0:0x1ac4]

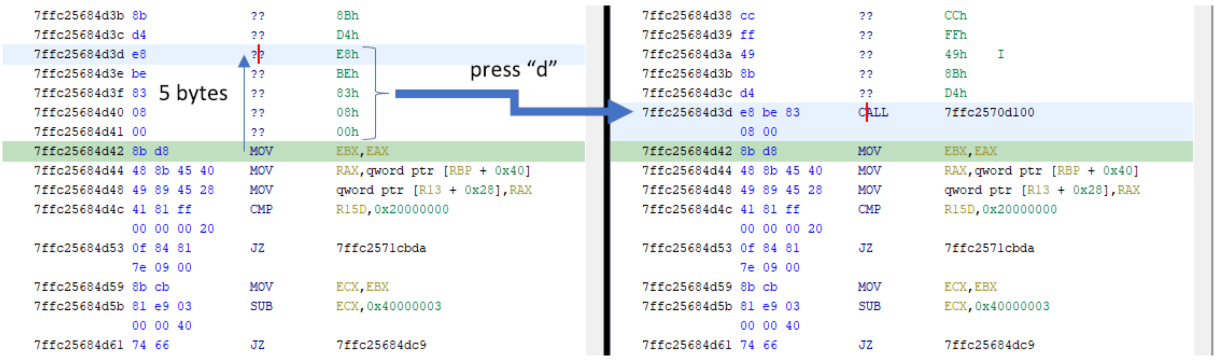
Level	PC	Function	Comment
0	7ffc256a2ad0		
1	7ffc25684ae3		
2	7ffc25684479		
3	7ffc256db1dd		
4	7ffc2568fb31		
5	7ffc256873e4		
6	7ffc25686af4		
7	7ffc257435ef		
8	7ffc256e4ceb		
9	7ffc256e4b73		
10	7ffc256e4b1e		

Figure 4.11: Stack Component

Clicking on one of the entries in this component will update the “Dynamic Listing” and “Registers” components to align with the selected stack frame. The registers are shown as `0` for all except for the current stack frame.

What is a bit confusing - because it’s inconsistent with how most IDEs do things - is that the “Dynamic Listing” view for non-current stack frames selects the instruction to which execution **will return**, not the `CALL` instruction which created the following stack frame.

If you’d like to see the call instruction which created the following stack frame, you can in most cases go back a number of bytes (depending on the type of `CALL` - instruction codes start with `E8`, `FF`, and in 32bit also `9A`) from the selected address and press lower case “D”. The `CALL` instruction will be disassembled.



```
7ffc25684d3b 8b              ??        8Bh
7ffc25684d3c d4              ??        D4h
7ffc25684d3d e8              ??        E8h
7ffc25684d3e be              ??        BEh
7ffc25684d3f 83              ??        83h
7ffc25684d40 08              ??        08h
7ffc25684d41 00              ??        00h
7ffc25684d42 8b d8           MOV       EBX,EAX
7ffc25684d44 48 8b 45 40     MOV       RAX,qword ptr [RBP + 0x40]
7ffc25684d48 49 89 45 28     MOV       qword ptr [R13 + 0x28],RAX
7ffc25684d4c 41 81 ff        CMP       R15D,0x20000000
             00 00 00 20
7ffc25684d53 0f 84 81        JZ        7ffc2571cbda
             7e 09 00
7ffc25684d59 8b cb           MOV       ECX,EBX
7ffc25684d5b 81 e9 03        SUB       ECX,0x40000003
             00 00 40
7ffc25684d61 74 66           JZ        7ffc25684dc9
```

```
7ffc25684d38 cc              ??        CCh
7ffc25684d39 ff              ??        FFh
7ffc25684d3a 49              ??        49h    I
7ffc25684d3b 8b              ??        8Bh
7ffc25684d3c d4              ??        D4h
7ffc25684d3d e8 be 83        CALL      7ffc2570d100
             08 00
7ffc25684d42 8b d8           MOV       EBX,EAX
7ffc25684d44 48 8b 45 40     MOV       RAX,qword ptr [RBP + 0x40]
7ffc25684d48 49 89 45 28     MOV       qword ptr [R13 + 0x28],RAX
7ffc25684d4c 41 81 ff        CMP       R15D,0x20000000
             00 00 00 20
7ffc25684d53 0f 84 81        JZ        7ffc2571cbda
             7e 09 00
7ffc25684d59 8b cb           MOV       ECX,EBX
7ffc25684d5b 81 e9 03        SUB       ECX,0x40000003
             00 00 40
7ffc25684d61 74 66           JZ        7ffc25684dc9
```

Figure 4.12: Disassemble `CALL` Instruction

Make sure in "File" > "Configure" > "Debugger" you’ve enabled "DebuggerStackPlugin"!

4.5 Program Analysis

Now that we know the different components of *Ghidra* we'll be dealing with, we can start debugging a program.

Now that you've gotten to know all the components we'll be using and have enabled them, I would suggest you start out with a layout such as the following (you can always taylor it to your needs):

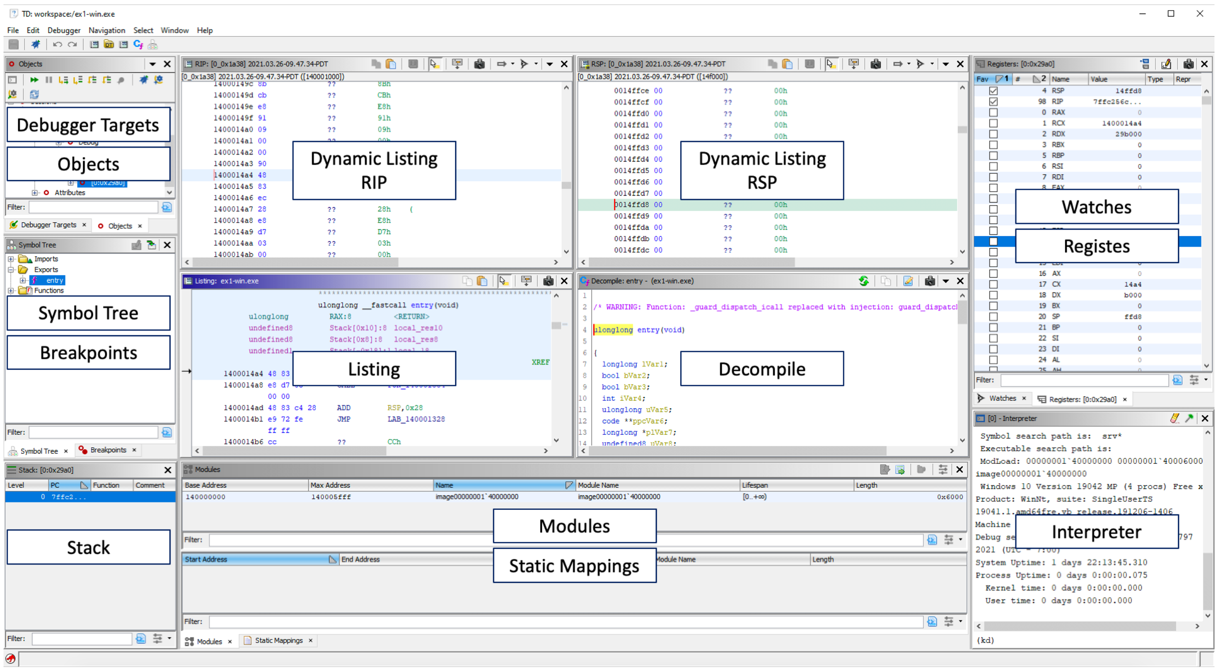


Figure 4.13: Debugger Layout

4.5.1 Running a Program

We've already seen how we can load a program into the debugger. The reason we don't use the convenient "Quick Launch" button (which runs the debugger connector in a separate JVM) is that unfortunately, it currently doesn't work with programs that take input parameters from the command line.

Time to exercise!

4.5.2 Suspending at a Known Location

In the "Static Analysis" chapter we looked at how to find the `main` method of a program in *Windows* - that's a good starting point (we're likely not very much interested in all the *Windows* internals that happen beforehand).

I've found the easiest thing to use the "Decompiler" component to navigate to the `main` method, then switch to the static "Listing" component and to set a breakpoint at the start of the method.

Once the breakpoint is set, we want the program to run up to that point. To get it to do so, we select the current session within the "Sessions" node and press the "Resume" (two green arrows pointing right) button in the "Objects" component. Alternatively, we can push the "F5" key (assuming standard key bindings).

When the debugger interrupts the execution, we have to check if we're really at our chosen breakpoint. If all goes well, *Ghidra* in the "Objects" component will mark the thread we're interested in (with the same gray arrow we've already seen in the "Listing" component). If this isn't the case, or to confirm we're really where we expect to be, we should check the "Dynamic Listing" component after clicking on the node representing the thread in the "Objects" component. Two alternatives to this approach are the following:

- check the "Listing" component and look for the arrow indicating the location of the *instruction pointer*.
- have a look at the "Interpreter" component. In *Windows*, it will state that it stopped at a breakpoint, e.g "Breakpoint 0 hit".

> When the program is suspended (and in fact each time there is a context switch: when a different thread gets to execute on a CPU), all the registers of the current thread are saved in the **thread context**, to be restored when it resumes. We will see later how we can edit information in this thread context, giving us full control of what is happening in the program.

Time to exercise!

4.5.3 Stepping

After getting to our starting point, we can now start navigating through the program. To do so, we have four options:

Step Kind	**Target**	**Comment**
Step Into	Next instruction. For `CALL`, go to first instruction of called function.	
Step Over	For `CALL`, step to instruction right after called function `RET`urns. Otherwise next instruction	
Step Finish	Instruction following the `RET`urn from the current function	Normally “Step Out”
Step Last	Next `CALL` or `RET` instruction	*Windows* only

> “Step Last” isn’t documented (yet), neither in *Ghidra*’s help, nor in the code. Reading the code, it turns out that it only exists for *Windows* (so far), where the debugger command executed by the connector is `tct`.

The buttons’ actions always relate to the thread currently selected in the “Objects” component.

Note that when stepping through multi-threaded or multi-process programs, another thread than the one we’re interested in may progress, so execution might be interrupted in another thread or process than the one we were operating on. In this case, it’s simplest to set a breakpoint at the next line we’re interested in and to “Resume” execution on the session node.

Time to exercise!

4.5.4 Modifying Register Values

We can actually ***change*** register values while debugging! If we want to do so, we need to enable editing registers by clicking the button “Enable editing of recorded register values” in the “Registers” component. Once that is enabled, double clicking on the cell in column “Value” will enable an input field in that cell.

Being able to modify a program’s state by modifying registers can be very handy if we need to e.g. change where a conditional jump goes. As we know that conditional jump instructions such as `JNZ` evaluate the flag registers (`ZF` in the case of `JNZ`), so changing the according flag registers will change the the outcome of the jump instruction.

But why would we want to do this? Say a program evaluates a passcode character by character and aborts if a character doesn’t match what it expects, by changing the control flow, we can get each expected characcter!

Time to exercise!

4.5.5 Modifying Memory

At the time of writing, there unfortunately isn't a convenient way to edit memory via the UI (I've opened an issue for that), so we'll have to use the "Interpreter" component to directly work with the native debugger.

One thing to be weary of is that most systems today write to memory in Little Endian [1] fashion. Let's have a look at a few examples showing what that means - specially note step 3! We will always write to the same memory location, e.g. `0x10`.

Step	Operation	Memory
0	Initial	`00 00 00 00 00 00 00 00`
1	Write one byte: `1f`	`00 00 00 00 00 00 00 1f`
2	Write a "word": `a2 8b`	`00 00 00 00 00 00 a2 8b`
3	Write `"foo"` into four characters	`00 00 00 00 00 6f 6f 66`
4	Write a "quad word": `ffeeddccbbaa9988`	`ff ee dd cc bb aa 99 88`
5	Write `2c` into a "double word"	`ff ee dd cc 00 00 00 2c`

4.5.5.1 GDB

In *UNIX*, working with *GDB*, we use the `set` command to write to memory. Specifically, we use `set {$size}$addr = $val`, where

- `$size` is a *C*-style type indicator specifying how much of the memory to overwrite. The most relevant examples are
  - `char` / `short` / `int` / `long` result in 1 / 2 / 4 / 8 bytes being written
  - `char[4]` writes four characters
- `$addr` is the address to start writing at.
- `$val` is the value to write to memory. This can be a string (in quotes) or a number (decimal / hexadecimal: prefixed with "0x").

Let's have at some examples, writing to memory address `0x10`:

Step	Operation	Memory
1	Write one byte: `1f`	`set {char}0x10 = 0x1f`
2	Write a "word": `a2 8b`	`set {short}0x10 = 0xa28b`
3	Write `"foo"` into four characters	`set {char[4]}0x10 = "foo"`
4	Write a "quad word": `ffeeddccbbaa9988`	`set {long}0x10 = 0xffeeddccbbaa9988`
5	Write `2c` into a "double word"	`set {int}0x10 = 0x2c`

[1] https://en.wikipedia.org/wiki/Endianness

4.5.5.2 WinDbg

In *Windows*, working with *WinDbg*, we can use the commands starting with `e` to overwrite memory areas. Here are the different versions you'll likely want to use:

- `eX $addr $vals`, where
  - `X` is
    - `b`, `w`, `d`, `q` to mean either **b**yte, **w**ord (2 bytes), **d**ouble word (4 bytes) or **q**ad word (8 bytes) values.
    - `f` or `D` to mean either single-precision **f**loating point (4 bytes) or **D**ouble precision floating point (8 bytes) values.
  - `$vals` is a space separated list of values to write.
- `eX $addr $str` or `ezX $addr $str`, where
  - `X` is either `a` to mean **a**scii or `u` to mean **u**nicode.
  - `$str` is a quoted string, which will be zero-terminated if the optional `z` was provided.

In both cases, `$addr` is the address to start writing at. Writing will continue as long as there are more entries in `$vals` or not all bytes in `$str` have been written. Let's have at the same examples, again writing to memory address `0x10`:

Step	Operation	Memory
1	Write one byte: `1f`	`eb 0x10 1f`
2	Write a "word": `a2 8b`	`ew 0x10 a28b`
3	Write `"foo"` into four characters	`eza 0x10 "foo"`
4	Write a "quad word": `ffeeddccbbaa9988`	`eq 0x10 ffeeddccbbaa9988`
5	Write `2c` into a "double word"	`ed 0x10 2c`

Time to exercise!

4.5.6 Finishing Up

When we've found out what we're interested in, we can stop our debugging session. This can be done by right-clicking the running process' node(s) in the "Objects" component ("Debugger" > "Sessions" > "$session > Processes > $process) and choosing"Kill". Be careful not to use "Detach" by accident, that will let the program continue running - which is not what you typically want.

Should the program not terminate, you can always terminate *Ghidra*. Since we've run the debugger in the same JVM, the process will die along with it.

4.6 Patching Programs

There's one more trick up *Ghidra*'s sleeve: we can directly patch programs, modifying their instructions and re-running them. This means that instead of changing register flags on the fly to change the result of e.g. a `JGT` command to not perform the jump, we can simply remove the jump instruction altogether! Or we can replace it by it's inverse, `JLE`.

To do so, we can right-click into the "Listing" component and choose "Patch Instruction". The first time around, we'll be given some warning / information we have to confirm. Then we'll be able to choose a new instruction to replace the existing one with.

To simply not execute an instruction, we can replace it with `NOP` (**No OP**eration - in bytes: `90`). Be careful to make sure you replace all of the bytes that make up the original instruction - if you don't, you'll get unexpected behavior! If you have two bytes to replace, you can use the two byte form of `NOP` (`66 90`) instead of writing two one-byte `NOP` instructions.

After changing the program, we need to export it to be able to execute it. We do this by selecting "File" > "Export Program", selecting "Format": "Binary" and confirming. The exported program will have a ".bin" extension. If we remove it, we will have an executable binary!

Make sure in "File" > "Configure" > "Ghidra Core" you've enabled "AssemblerPlugin" and "ExporterPlugin"!

Time to exercise!