GETTING STARTED WITH GHIDRA: THE SOFTWARE REVERSE ENGINEERING TOOL FROM THE NSA

By Clara Andress – ISSA member, Puget Sound Chapter
and Jason Andress – ISSA Senior Member, Puget Sound Chapter

This article discusses software reverse engineering and the specific use of the Ghidra tool released by the NSA. Specifically covered are a walkthrough of the tool and the use of it to recover the password included in an example compiled binary.

Abstract

One of the best ways to figure out how something works is to take it apart and see its internal workings and how everything fits together. We can then use this information to analyze the target and how it was designed, allowing us to document, reproduce, alter, or just interact more deeply or efficiently with it—this process is known as reverse engineering. The article discusses reverse engineering tool Ghidra.

Reverse engineering has a long history including industrial espionage and competitive intelligence efforts between companies or nations, hackers and makers modifying or altering hardware devices and software applications, software pirates defeating digital rights management (DRM) controls on software and contents, and security researchers taking software or hardware apart to find flaws in it. Reverse engineering is a very common task across many of the industries that focus on or interact with technology in some fashion.

In the world of security research, many (but not all) of our reverse engineering efforts focus on software or on the firmware that is behind the interfaces of a hardware device. Fortunately, with software reverse engineering, having a bag of parts left at the end isn't a problem as we don't even have to be able to put it back together again. Software reverse engineering (SRE) tools allow us to decompile applications to get something like the original source code, or a reasonable approximation thereof, back out of them and see how they work. This type of reverse engineering is often binary reverse engineering, as we are working with compiled programs (called binaries), and do not have access to the original source code.

The tool set for software reverse engineering includes a variety of arcane-sounding utilities such as debuggers, hex editors, disassemblers, decompilers, and many other small utilities that allow us to take a peek inside our target application and see what it's doing. This may sound like absolute black magic to the uninitiated, but it really isn't all that difficult or complicated with the right tools in hand, at least at a basic level. In fact, we're going to be doing some of this shortly. For the moment, back to Ghidra.

Ghidra

Ghidra is a software reverse engineering tool developed by the US National Security Agency (NSA) that was released to the public at RSA's March 2019 conference [2]. Although it

wasn't yet available to the public, we did learn about the existence of this tool from WikiLeaks [3] in March of 2017.

The release of this tool is exciting for several reasons. Although there are many existing SREs, such as IDA Pro[1] and Binary Ninja,[2] they can be very expensive or lack or charge more for the full feature set. Ghidra, on the other hand, is a free, open source, full-featured, and multi-platform SRE platform. Some long-time users of other competing tools may or may not be fans and quibble over certain features being missing or different, but it is, overall, a very impressive tool for being free and open source.

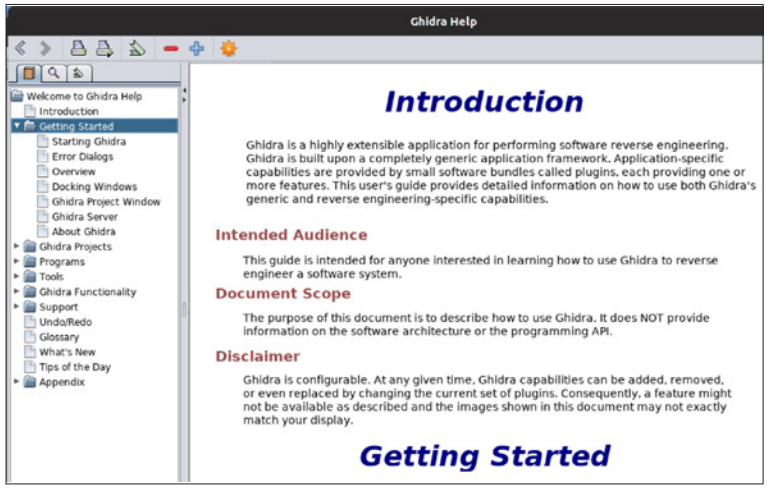

Figure 1 - The Ghidra help documentation

Ghidra can decompile, disassemble, and assemble a program. It has a code browser, graph viewer as well as code patching, search, and diffing tools. It is scriptable and has an undo/redo feature (this is surprisingly uncommon). We can easily change our user experience; it can be run from the command line or graphical user interface and can even be themed. Since it is open source, a community has developed around it and is already busily building new plugins and features via the conveniently exposed API.

Some in the security community may look askance at a tool from the NSA from under the protection of their tinfoil hats; however, since Ghidra is open source, the code is available for review. Ghidra has been pored over by many security researchers and interested partied outside the NSA. After being released for only a few hours a number of vulnerabilities were uncovered, which were resolved in the next release [1]. There has been, at this point, no smoking gun backdoor that would indicate anything untoward in the code.

A brief tour of Ghidra

Before we jump into the thick of things, let's take a quick look at Ghidra, which has a fairly solid set of documentation. Online, we can find a FAQ on the NSA's Github repository for Ghidra.[3] The tool also has extensive built-in documentation, which can be accessed within the tool itself by pressing F1 (figure 1).

Ghidra is an enormous tool, full of features, many of which are well beyond the scope of this brief discussion. Let's quickly talk about some of the main parts of the interface that we

1 "IDA: About," Hex-Rays – https://www.hex-rays.com/products/ida/.

2 "Binary Ninja: A New Kind of Reversing Platform," Binary Ninja – https://binary.ninja/.

3 NationalSecurityAgency/ghidra, "Frequently Asked Questions," Github – https://github.com/NationalSecurityAgency/ghidra/wiki/Frequently-asked-questions.

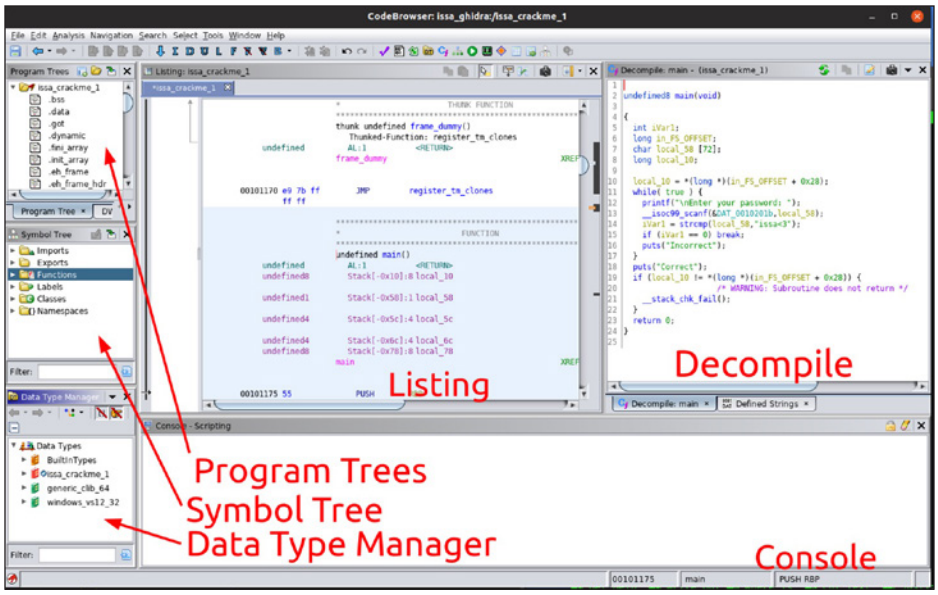

Figure 2 - The Ghidra code browser

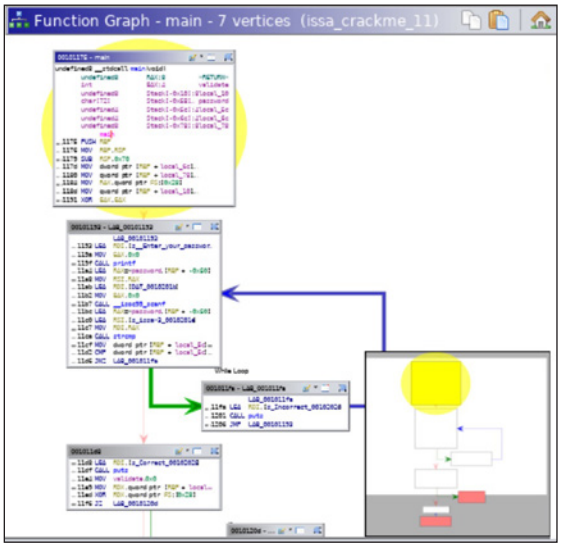

Figure 3 - The function graph

will be working with as we tackle our example program. The main windows that appear in the interface when launched are *Program Trees, Symbol Trees, Data Type Manager, Listing, Decompile*, and *Console* (figure 2). There are several other windows such as the *Function Graph* in figure 3 that can be viewed by clicking on the Window menu at the top.

Program trees can be used to organize the disassembly of the program (as shown in the *Listing*) in a variety of different ways. *Symbol Trees* shows all of the symbols (addresses of its variables and functions) in the program, such as imports/exports and functions. *Data Type Manager* shows the defined data types in use by our program, as well as those in libraries or in Ghidra. *Listing* shows the machine language disassembly of the program. This listing is populated by default when we import a binary. *Decompile* shows the decompiled version of the program and is synchronized with the *Listing* window. *Console* is used to show the output of running programs within the tool itself. *Function Graph* shows the flow of the program code in a graphical manner.

Ghidra "Hello World"

Let's take a look at a very simple task that we can carry out with Ghidra. This won't require any knowledge of assembly language or reverse engineering. On the other hand, we're about to compile some code from scratch and then use a decompiler to get the code back so we can find a secret password!

First, download and install Ghidra.[4] The installation process is relatively simple and well documented on the site,[5] so it shouldn't be problematic. Ghidra can run on Windows, Linux, and macOS platforms, and we'll be using Linux in the examples here. To follow along with this on another platform, the binary will need to be compiled for that platform in order to run it.

Next, download the binary file *issa_crackme_1* that we'll be working with from the GitHub repository.[6] Either browse to the website and download it directly or download the entire repository by running the command *git clone https://github.com/jandress/issa_ghidra*. If cloning the repository, the command will automatically place the contents into a new directory called *issa_ghidra*; if downloading it manually, create this directory and place the files into it.

This type of file and associated reverse engineering exercise is often referred to as a "crackme," a small program used to test or learn reverse engineering. These are often part of a series, increasing in complexity and difficulty as they progress.

If you are on Linux, the binary file *issa_crackme_1* from the repository will be ready to use directly, or the code can be compiled with the GNU Compiler Collection (gcc) compiler by running the command *gcc -g -o issa_crackme_1 issa_crackme_1.c*. This process should also work on macOS but will be slightly different for Windows[7]. For both Windows and macOS, the code will need to be compiled in order to be able to execute the program. Compiling the code is not a requirement if we just want to look at it in Ghidra and not actually run the crackme binary.

Binary in hand, we can run the program. On Linux, this can be done by executing *./issa_crackme_1* from the issa_ghidra directory. As the crackme runs, it will prompt for a password,

4 Ghidra – https://ghidra-sre.org/.

5 "Ghidra Installation Guide" – https://ghidra-sre.org/InstallationGuide.html.

6 https://github.com/jandress/issa_ghidra.

7 "Walkthrough: Compile a C Program on the Command Line," Microsoft — https://docs.microsoft.com/en-us/cpp/build/walkthrough-compile-a-c-program-on-the-command-line?view=vs-2019.

which, unless we have peeked at the source code, we will not have as seen in figure 4.

```
Enter your password: hello
Incorrect

Enter your password: 1234
Incorrect
```

Figure 4 - The issa_crackme_1 program executing

Exit the program and let's see what we can do about getting the password.

Firing up Ghidra

In the directory where Ghidra was unpacked during the install (currently ghidra_9.0.4), execute *./ghidraRun* on Linux or macOS, or *ghidraRun.bat* on Windows. The Ghidra splash screen will briefly appear; then the project dialog window will display. Here, we will select File, New Project, Non-Shared project (shared allows for collaboration with others) and browse to the issa_ghidra directory we created earlier, also setting the project name to *issa_ghidra*. Lastly, click the green dragon head on the upper left to launch the code browser.

Now we need to load our crackme binary into the project by clicking File, Import File, and choosing the *issa_crackme_1* file from our project directory, then clicking Select File To Import. On the Import dialog, keep the defaults and click OK.

We should now see something like figure 5, the Import Results Summary and a prompt asking to analyze the file, to which we should click *Yes*. On the Analysis Options dialog, keep the defaults and click Analyze; then click *OK* to close the Import Results Summary.

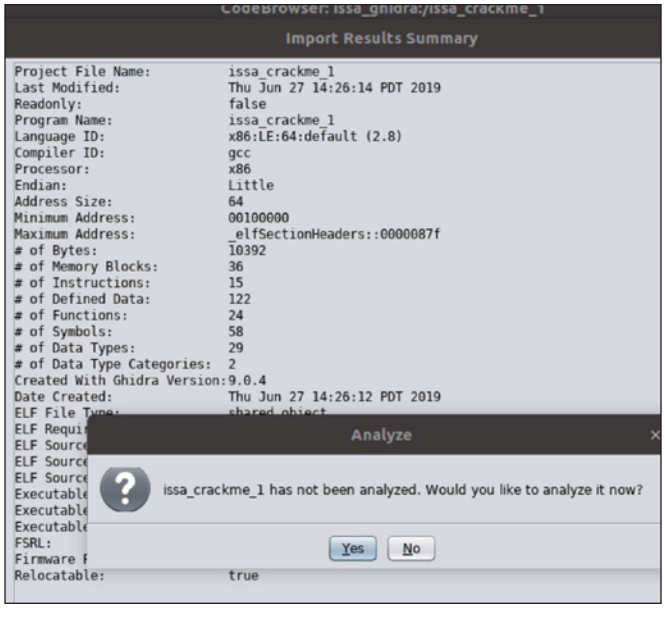

Figure 5 - The Import Results Summary and Analyze Prompt

That was the hard part. Now let's see what we can see about the crackme using Ghidra.

Taking a look around

We can see that the windows in the Code Browser are presently largely blank. In order to see interesting things, we need to pick a function out of the Symbol Tree window on the middle left. Expand the Functions directory, then scroll down and click on the main function - we can now see the decompiled version of our crackme in the Decompile window on the right, as shown in figure 6. The variable names shown may be slightly different than the screenshot here.

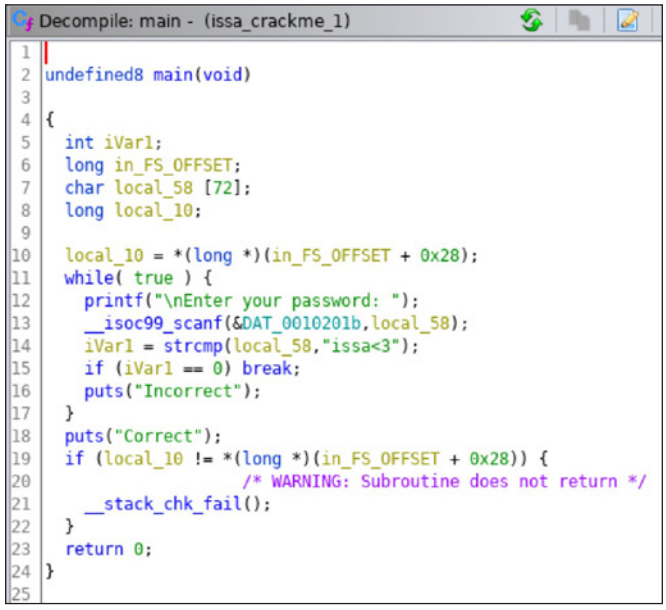

Figure 6 - The Initial Decompile of issa_crackme_1

Interestingly enough, without taking any other steps, we can see what might be a password by looking at the strcmp (this is the C function used to compare two strings) on line 14 in figure 7.

```
14    iVar1 = strcmp(local_58,"issa<3");
```

Figure 7 - The password?

Cleaning up the decompiled code

In order to make the decompiled code more legible, we can clean things up a bit by renaming the variables to make things more clear. For purposes of this example, we'll only make a few simple changes.

Right-click on the name of the char variable, local_58 in the example, and choose *rename variable*, then change the name to *password*. The automatic variable names will shift around again after doing so. Now do the same with the *int* and rename it *validate*. Due to the simplicity of the code that we are working with, this is relatively straightforward to puzzle out; we have one int (integer) and one char (string) variable in our code and we know what they were originally called because we have the original source. This will, of course, be much more complex in real-world applications; however, this is a *hello world* exercise and intentionally made simple.

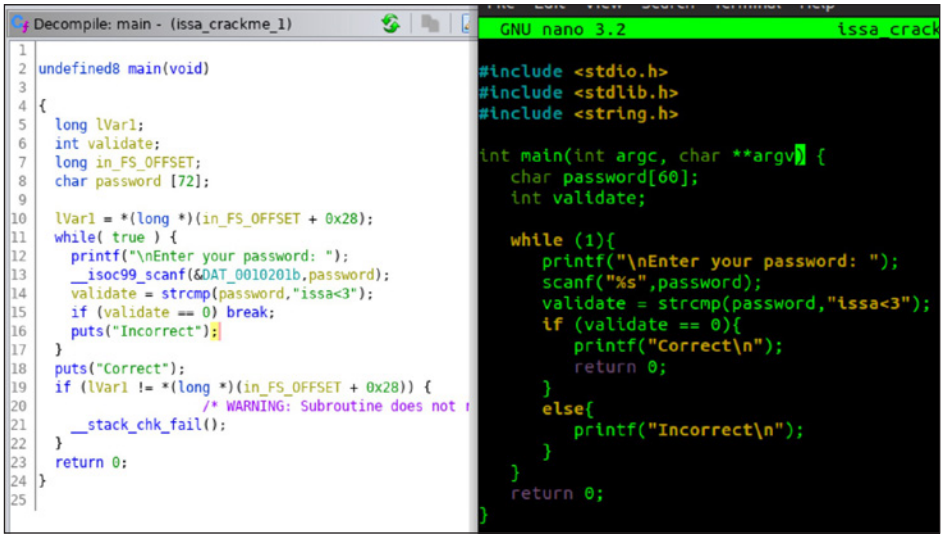

Figure 8 - Comparison of decompiled and original code

If we look at the side-by-side comparison in figure 8, there are a few differences yet, and we can see where the compiler inverted the implementation for our password check logic for efficiency, but this looks pretty good for decompiled code from the binary of our crackme.

Also worth noting is that we could have arrived at the same information, or at least a reasonable guess at it, by looking at the *Defined Strings* window, where browsing through the strings would have revealed something password-ish, as shown in Figure 9.

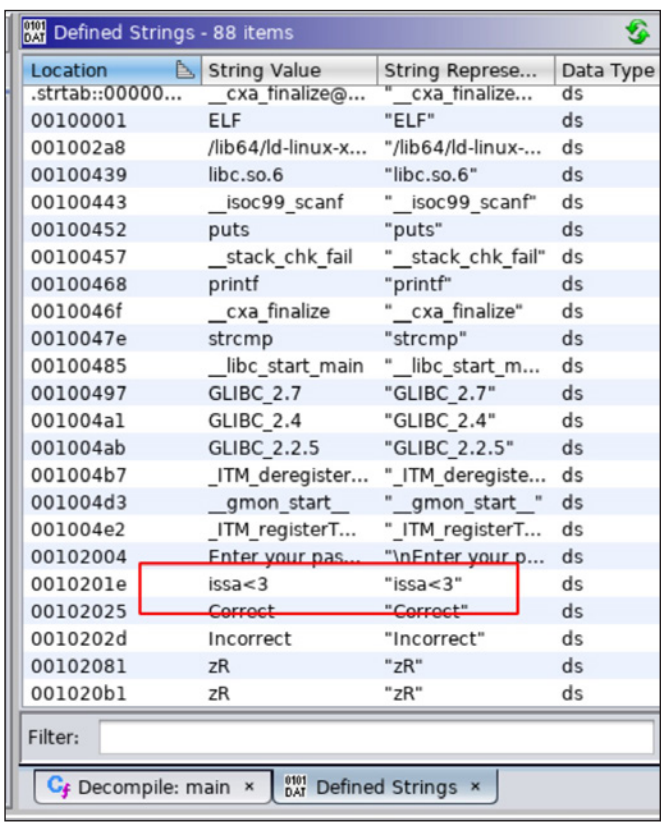

Figure 9 - The Defined Strings window

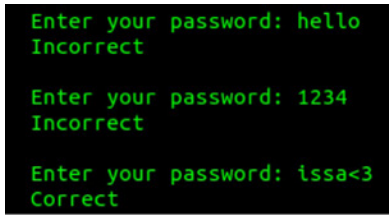

Figure 10 - Success!

Testing again

Now that we have the password, let's run the program again by executing *./ghidraRun* on Linux or macOS or *ghidraRun.bat* on Windows. This time we can enter the correct password, *issa<3*, and we should see the "Correct" result, as seen in figure 10.

Arguably, we could have arrived at this result by running the strings tool like *strings issa_crackme_1* on Linux and arrived at the same result, as seen in figure 11, although we wouldn't have had the excuse to play with Ghidra. This is, however, an excellent reminder that it is sometimes, but not always, better to work smarter, rather than harder.

Defeating software reverse engineering

The sample application that we were working with here was intentionally a "happy path" one for purposes of this example—it was constructed and compiled in such a way as to be easily reversed. This will clearly not always be the case. There are a number of controls that we can put in place in order to hinder reverse engineering efforts on applications, varying from the simple to the highly complex. In general, the more effort that we put into making applications resistant to re-

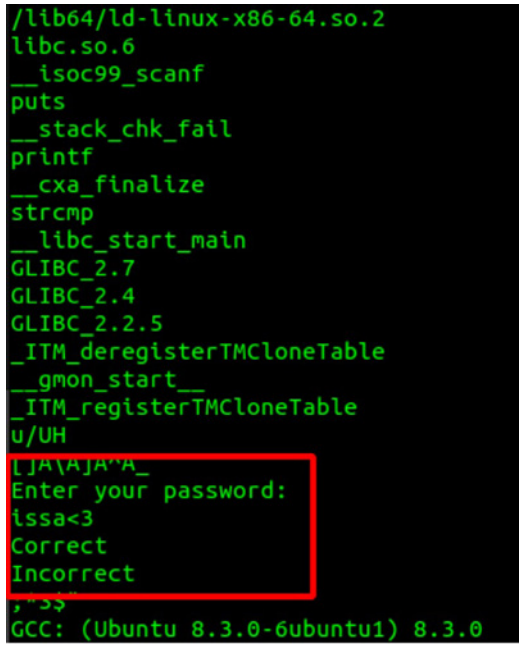

Figure 11 - strings command output

verse engineering, the more expensive, in terms of resources and efficiency, our application will be when it executes. Let's quickly look at just a few of the options that we might use here.

Choosing the properly secure compile settings is one of the first steps to defeating SRE. Our example above was compiled in the most "chatty" way possible, in order to provide Ghidra more material to work with when taking it apart. These hardening settings will vary according to the language and compiler being used, but most modern compilers have a particular set of flags to automatically harden the compiled output. A quick trip through Google searching for the name of the compiler and "hardening" will turn up the vendor documentation for this.

Obfuscation is another simple method that will help defeat SRE. In our example above, we could easily see the strings in the application, including our password in plaintext. A simple obfuscation effort would be to encrypt or encode out the password, which would defeat our above efforts entirely without anything impactful in the way of additional resources. We can go much further with this concept by including dummy code that doesn't ever get called or has no real functionality in our application when it does. Every additional bit of functionality, variable, call, etc. that we add to our application makes it more complex for both the tool and the person operating it to see what is going on under the hood.

We can also resort to larger-scale efforts to hinder reverse engineering, such as encryption and packing. Just as these sound like, these efforts involve wrapping up portions of the application inside packages that are encrypted and/or compressed. When the application executes, a loader unpacks those sections of the program and executes them. This is a very common tactic used in malware and fairly well-known among those who reverse engineer malware samples. Depending on how exactly this is implemented, it can provide varying degrees of protection against reverse engineering.

Environmental awareness is an interesting method for defeating reverse engineering. This type of protection attempts to discern what environment it is running in and alters its behavior based on this detection. This is another commonly used technique by malware authors and is commonly used to detect if the application is running in a virtual machine, sandbox application, or debugger. Once such an environment is detected, the application may refuse to run at all, or may execute a different set of functions entirely in order to frustrate reverse engineering efforts. This type of protection can be very effective as it directly targets the tools that would be used to defeat it.

These are just a few of the wide variety of options that are available to developers to protect their applications, and only a surface view, at that. There is an entire specialization of the security field that deals with such efforts, and the topic is a very deep one. One of the main driving forces behind such efforts are the video game and media industries, both of which fight an ongoing battle to secure their content.

Conclusion

Ghidra, being open source, has grown a community of dedicated security professionals who are contributing to the project and enjoys active security community discussion on social media sites such as Reddit[8] and Twitter.[9] In a recent Google CTF, Ghidra also featured strongly as a choice of tool among several of the highly-skilled finishers of the challenge.[10] With its attractive price tag, growing set of features, and support of the security community it is a very viable and promising software reverse engineering tool.

References

1. Hashim, A. 2019. "Critical Vulnerabilities Found in Recently Released NSA Reverse Engineering Tool 'Ghidra,' " Latest Hacking News – https://latesthackingnews.com/2019/03/24/critical-vulnerabilities-found-in-recently-released-nsa-reverse-engineering-tool-ghidra/
2. Joyce, R. 2019. "Get Your Free NSA Reverse Engineering Tool,"SA USA 2019 – https://published-prd.lanyonevents.com/published/rsaus19/sessionsFiles/13678/PNG-T09-Come-Get-Your-Free-NSA-Reverse-Engineering-Tool%21.pdf.
3. Wikileaks. 2017. "Vault 7: CIA Hacking Tools Revealed" – https://wikileaks.org/ciav7p1/cms/page_9536070.html.

About the Authors

Clara Andress is an application security expert, with a strong background in development and operations. She is a recovering government contractor and enjoys wearing many and varied hats. She may be contacted at clara.a.andress@gmail.com.

Dr. Jason Andress is a seasoned security professional, security researcher, and technophile. He has been writing on security topics for over a decade, covering data security, network security, hardware security, penetration testing, and digital forensics, among others. He may be reached at jason.andress@gmail.com.

8 "/r/ReverseEngineering," Reddit.com – https://www.reddit.com/r/ReverseEngineering/.

9 @GHIDRA_RE, "Ghidra," Twitter.com – https://twitter.com/ghidra_re.

10 John Hammond and LiveOverflow, "Reverse Engineering Race Feat," YouTube – https://www.youtube.com/watch?v=5G9KjZ62X_U.