ARL-TR-8986 ● JUNE 2020

DEVCOM ARMY RESEARCH LABORATORY

Hands-on Cybersecurity Studies: Uncovering and Decoding Malware Communications with Dshell

by Daniel E Krych and Jaime C Acosta

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8986 ● JUNE 2020

Hands-on Cybersecurity Studies: Uncovering and Decoding Malware Communications with Dshell

Daniel E Krych and Jaime C Acosta
Computational and Information Sciences Directorate,
CCDC Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2020	Technical Report	May 2019–March 2020

4. TITLE AND SUBTITLE
Hands-on Cybersecurity Studies: Uncovering and Decoding Malware Communications with Dshell

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Daniel E Krych and Jaime C Acosta

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CCDC Army Research Laboratory
ATTN: FCDD-RLC-ND
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8986

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
ORCID ID: Jaime C Acosta, 0000-0003-2555-9989

14. ABSTRACT

This report presents a hands-on exercise on basic software reverse-engineering with the ultimate objective of learning the way that a particular malware (malicious software) is communicating across a network, and developing software to detect and reveal these communications in plaintext, in vivo. Remote access trojans (RAT) are a type of malware that persist on the infected machine ("bot") after compromise and provide the malicious actor in control of the malware with remote access to the infected machine via established command-and-control channels. As with all malware, RATs are typically spread through phishing emails or websites where the software is downloaded without the user knowing; it can also spread by taking advantage of vulnerabilities in software running on the victim's devices. This report details the last of three software reverse-engineering exercises, which can be completed cumulatively or individually as each accomplishes a specific task and builds off the previous exercise. The effects and communications of RATs are demonstrated and participants are guided through a series of steps leveraging the US Army Combat Capabilities Development Command (CCDC) Army Research Laboratory's (ARL's) open-sourced network forensic analysis framework, Dshell, to develop a "Dshell decoder" that can decode the communications to enable detection and support mitigation.

15. SUBJECT TERMS
Dshell, traffic analysis, decode, software reverse-engineering, trojan, remote access trojan, RAT, malware, command-and-control, C2, hands-on cybersecurity, CyberRIG

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel E Krych
Unclassified	Unclassified	Unclassified	UU	22	19b. TELEPHONE NUMBER (Include area code) (301) 394-1582

Contents

List of Figures

1. Introduction

This report presents a hands-on exercise on basic software reverse-engineering with the ultimate objective of learning the way that a particular malware (malicious software) is communicating across a network, and developing software to detect and reveal these communications in plaintext, in vivo.

Remote access trojans (RATs) are a type of malware that persist on the infected machine ("bot") after compromise and provide the malicious actor in control of the malware with remote access to the infected machine via established command-and-control (C2) channels. As with all malware, RATs are typically spread through phishing emails or websites where the software is downloaded without the user knowing; it can also spread by taking advantage of vulnerabilities in software running on the victim's devices. This report details the last of three software reverse-engineering exercises, which can be completed cumulatively or individually as each accomplishes a specific task and builds off the previous exercise. These exercises and their reports demonstrate the effects and communications of RATs and guide participants through a series of steps to uncover, analyze, and develop software to detect the malware.[1,2]

In the first software reverse-engineering exercise, Wireshark, a network protocol analyzer, is used to analyze the malware traffic between the C2 server and the bot, and Volatility, a memory forensics tool, is used to analyze an image of the infected hard drive's memory (memory dump) and find and extract the malicious program (binary).[3,4] In the second exercise, Ghidra, the National Security Agency's open-source software reverse-engineering framework, is used to reverse-engineer the communications between the C2 server and the bot by analyzing the malicious binary.[5] This report details the third and final software reverse-engineering exercise, which walks the participant through a series of steps to leverage the US Army Combat Capabilities Development Command (CCDC) Army Research Laboratory's (ARL's) open-source network forensic analysis framework, Dshell, to develop a "Dshell decoder" to decode the RAT malware communications, which will enable detection and support mitigation.[6] Dshell is a powerful open-source network forensic analysis tool written in Python. It enables analysts to quickly and efficiently decode network data and it comes prepackaged with many decoders. It is easily extensible due to its modular design.

2. Setup and Configuration

The reverse-engineering hands-on exercises consist of three virtual machines (VMs): one is used as the C2 server, one is used as the infected machine (bot), and

one is used as the Analysis VM, which is placed in-between the C2 and bot machines with a promiscuous port, allowing it to see all traffic between the C2 and bot machines. This setup is seen in Section 5. Participants use the Analysis VM throughout these exercises to analyze malware traffic between the machines, extract the malware from the hard disk and analyze the memory dump, reverse engineer the communications by analyzing the malware binary, and finally develop software to detect and reveal these communications in plaintext.

The setup configuration consists of the following software elements:

- VirtualBox[7] (Version 6.0)
- Two Windows 7 Home Basic 32-bit VMs[8]
- One Ubuntu 18.04 LTS Linux 64-bit VM[9]
- Wireshark[3] (Version 3.0)
- Volatility[4] (Version 2.1)
- Ghidra[5] (Version 9.1.2)
- Dshell[6] (Python 2)
- ArchDeus (a set of scripts which mimic communications notional to RATs)

The Analysis VM was set up as an Ubuntu Linux machine with Wireshark, Volatility, Ghidra, and Dshell installed. The C2 machine was set up with scripts to communicate commands to the bot machine. A promiscuous port was set up to allow the Analysis VM to view all of the traffic between the C2 and bot machines.

The entire exercise runs on the CCDC Army Research Laboratory South Cyber Rapid Innovation Group (CyberRIG) Collaborative Innovation Testbed, which provides an isolated environment, ensuring that all the environmental artifacts are segregated from any real systems. Participants access the Analysis VM via a web-based interface to allow any system with a web browser to be used, and to isolate the exercise and its contents from the participants' machines.

3. Learning Objectives

This exercise teaches participants the following:

- Participants gain a better understating of how RATs work, which entails malicious actors using C2 channels to remotely control infected machines. The effects of the malware on the sandbox environment should emphasize

the importance of securing computer systems, and detecting and preventing malware.

- Participants gain experience in collecting network traffic and extracting necessary data by examining the data with Wireshark and Dshell.
- Participants learn about obfuscation through encryption, seeing the simple ROT cipher in action and how it turns human-readable text into gibberish, and how this can be reversed back to plaintext. This will teach participants about encryption, and how the same techniques they are using on this simple, but educational, ROT cipher could be applied to analyze and decode a highly complex, encrypted, and obfuscated piece of malware.
- Participants learn about the network forensic analysis tool Dshell, what it is capable of, how to use it to analyze network traffic for threat hunting, and most importantly, how to develop a custom Dshell decoder to decode the unique malware protocol communications used by this RAT. This will provide the building blocks for using Dshell as an analysis tool. It will also teach participants how they can develop custom decoders to decode other network protocols, or add logic to or modify existing logic in decoders to address their unique analysis needs and improve their overall security stance.

4. Methodology

In creating these software reverse-engineering exercises, we started with the desire to provide a hands-on learning approach to the development of a Dshell malware decoder. By leveraging multiple analysis tools and techniques to analyze the malware and its communications, we provide a way for participants to learn and practice forensic techniques and gain enough knowledge about the malicious binary and its actions to understand its inner workings and develop a uniquely crafted Dshell decoder, which enables detection and supports mitigation.

While developing these exercises, we intended for them to be educational for participants with varying levels of experience in network security, forensics, and programming. We chose to create scripts that mimic C2 communications notional to RATs and leverage a simple rotational cipher to encrypt the data, which provide an approachable, but still educational, malware binary to uncover, analyze, and decode. Novices will learn a breadth of knowledge and be able to step through the exercise instructions, understand the overall scenario, and develop the basic Dshell decoder. Experts will move more quickly through the exercises, but still learn new tools and analysis techniques, and can aim for going above and beyond in analysis, forensics, practicing using these tools, and in developing an efficient and effective

Dshell decoder. We purposely used simple encryption and code so all participants can follow along, but more-versed participants will notice the applicability of these same techniques for analyzing and decoding highly complex, encrypted, and obfuscated malware. This exercise could be modified to use a more complex encryption method or malware to increase its difficulty and provide more-advanced educational material.

In creating this exercise, we began by downloading the latest version of Dshell[6] (Python 2) from GitHub[*] onto the Ubuntu 18.04 LTS Linux 64-bit virtual machine. Dshell is CCDC Army Research Laboratory's popular network forensics tool released publicly in 2014, which enables the development of, and includes plugins, aka "decoders" for, decoding network communication protocols.[6] The ability for Dshell decoders to be rapidly prototyped and modified on the fly using the powerful, but user-friendly Python programming language makes it very useful for decoding malware, which commonly obfuscates communication and has slightly changed variants.

To get Dshell and the libraries it depends on installed and running on Ubuntu 18.04 LTS Linux, and drop into the Dshell shell environment (noted by the '**Dshell >**' prompt), we read the Dshell GitHub README and conducted the following steps in the terminal, within the user's home directory.[†]

```
git clone https://github.com/USArmyResearchLab/Dshell.git
./install-ubuntu.py
./dshell
```

We ran Dshell as a normal user since root permissions are not needed, and ensured the user had read permissions for the packet capture (pcap) files created and used with Dshell. For the best Dshell end-user experience, we ran Dshell by dropping into its shell environment; within it, normal UNIX commands can still be executed.

One important feature to note with Dshell is that if a pcap begins in the middle of a Transmission Control Protocol (TCP) connection, and thus the TCP handshake that occurs at the start of a connection is missing, Dshell will by default ignore that specific, incomplete connection unless the ***--decodername*_ignore_handshake** flag is set when running the decoder to ignore the handshake. For example, to ignore the TCP handshake when using the **netflow** decoder on a pcap file, run the following command:

decode -d netflow --netflow_ignore_handshake *pcapfilename*.pcap

[*] https://github.com/USArmyResearchLab/Dshell

[†] Note: the **install-ubuntu.py** Python program works on Ubuntu and scripts the README's steps of installing the Python libraries Dshell depends on, and runs the makefile.

5. Exercise

5.1 Mission Briefing

The briefing for the software reverse-engineering exercise is as follows:

You are a member of the special task force Weltall-42. You have been charged with uncovering a new and deadly RAT known as ArchDeus. Your team was able to 1) recover an infected hard drive with the infection and 2) place yourselves between a malware "command and control" and a "bot" (or victim machine) as seen in Fig. 1.

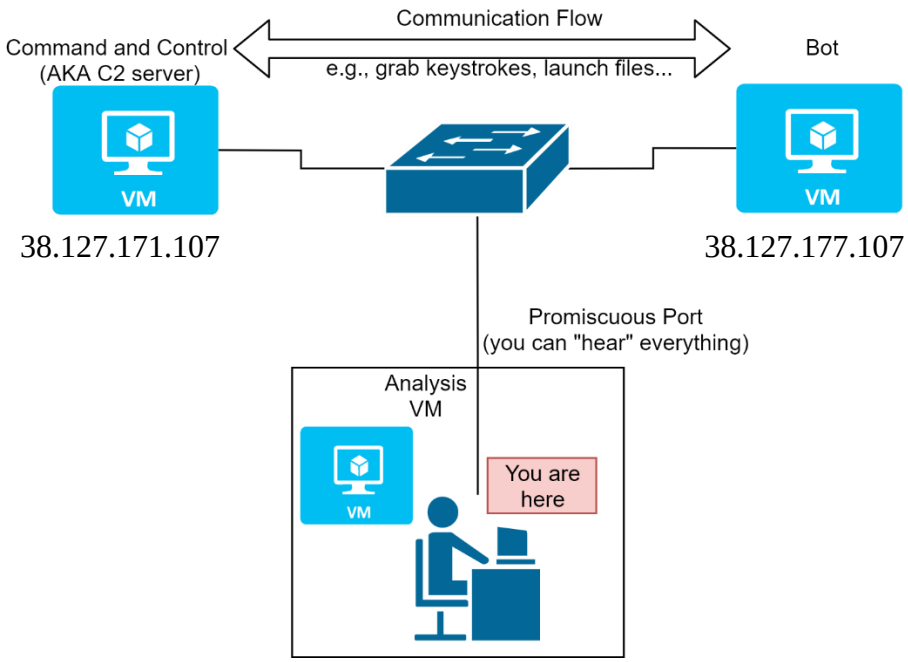

Fig. 1 Reverse-engineering exercise network scenario

The overall software reverse-engineering exercise series is separated into three main exercises, which build off each other to determine key information but can also be stand-alone exercises. This report covers the third exercise.

1) a. Analyze malware traffic between the C2 server and the Bot using a tool called Wireshark.

   b. Extract malware from a hard disk by analyzing the hard drive and pulling out the infected process using a tool called Volatility.

2) Reverse-engineer communications between the C2 server and Bot by analyzing the malware with a tool called Ghidra.

3) Develop a decoder for the malware traffic (Detection and Decoding Mechanism) using a tool called Dshell.

This exercise requires about 1.5–2 h to complete.

5.2 The Final Battle: Develop the Detection and Decoding Mechanism

Apply your skills to build tools to help decipher and defeat the malware.

1) The first thing we want to do is capture traffic that we can work with—this way we will have a static (nonchanging) data set and repeatable processes. Start a terminal **(Ctrl+Alt+t)** and then open Wireshark by running the following in the terminal.

   sudo wireshark

   (Enter the password **toor** when prompted.)

2) As shown in Fig. 2, double-click on the **enp0s3** label to start capturing traffic. Wait between 1–2 min and then press the stop button.

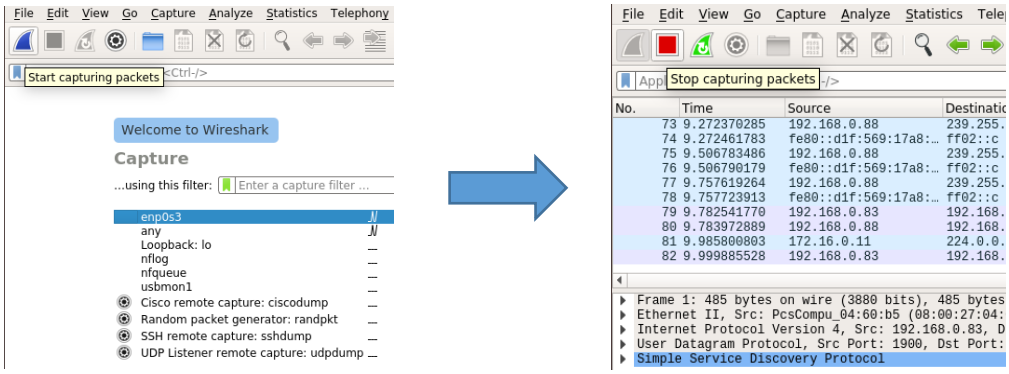

Fig. 2 Capture network traffic with Wireshark

3) Save your capture by clicking on **File->Save As…** and then select the ***/home/student/Dshell/*** directory. Name your file **mycap**, use the default file type of **pcapng**, and click the **Save** button. See Fig. 3 as an example.

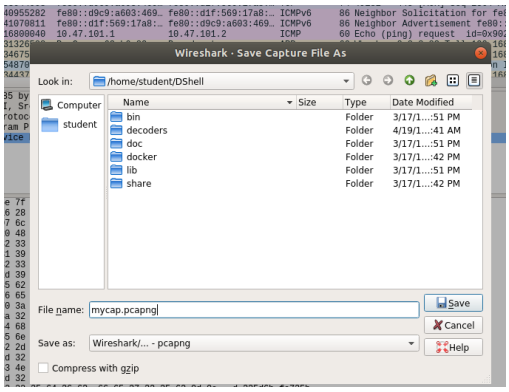

Fig. 3 Saving network traffic (pcapng) file captured with Wireshark

****

Dshell is a powerful open-source network forensic analysis tool written in Python. It enables analysts to quickly and efficiently decode network data and it comes prepackaged with many decoders. It is easily extensible due to its modular design.

****

4) Close Wireshark. Now you will use Dshell to build a decoder for the communication traffic. Open a terminal and look at run options and the available decoders by following these commands:

   cd /home/student/Dshell
   sudo chmod 644 mycap.pcapng *(updates permissions on the pcap file so all users can read it, and only the owner can write)*
   ./dshell *(you will now drop into the Dshell shell)*
   decode *(shows the available run options with descriptions)*
   decode -l *(lowercase 'L', shows the available decoders)*

5) Now run the prepackaged ***ip*** and ***dhcp*** decoders on the traffic you captured earlier. *(Note: the **dhcp** decoder may not return anything; that's okay.)*

   decode -d ip mycap.pcapng
   decode -d dhcp mycap.pcapng

   Try running a few other decoders on the traffic you captured, such as ***nbns***, ***netflow***, and ***followstream*** to learn more about the traffic data. You can also try running decoders on other pcaps within the Dshell folder.

6) Notice that each of these filter out network packets and display certain packet data. Now use the **ip decoder** and apply an **additional filter**. Run Dshell again, but change the default filter (called a Berkeley Packet Filter [BPF]) so that you only show User Datagram Protocol (UDP) traffic using port 67.

*(Note: This custom BPF filter is actually the default filter of the **dhcp** decoder, so if the **dhcp** decoder in 5) did not return anything, this will not either. The **dhcp** decoder first filters on the traffic, then decodes the DHCP protocol, breaking up the data into relevant fields and outputting it for an analyst to quickly gather information.)*

decode -d ip --bpf "udp and port 67" mycap.pcapng
(notice the 2 "minuses" before bpf)

7) Run your own command (will be similar to the previous question) to figure out what two IP addresses are communicating using TCP and port 9999. If you do not see any results, it may be that you did not capture enough traffic. *(Hint: in the output, you will see that addresses/port numbers are indicated as follows <IP Address>:<port>.)*

   a. Your command:

   _____

   b. Identified IP addresses: ___.___.___.___ and ___.___.___.___

8) Now we will write our own decoder. First, create a new directory for your plugin/decoder and then copy over a template (specifically for TCP traffic) that comes with Dshell. *(Note: all of the following commands can be run while still within the Dshell shell [Dshell>], but feel free to exit the shell by hitting ctrl+d or typing exit and hitting enter, and then you can start the shell again with **./dshell** within the /home/student/Dshell directory when needed.)*

cd /home/student/Dshell
mkdir decoders/challengeDecoders/
cp decoders/templates/SessionDecoder.py
decoders/challengeDecoders/archdeusSession.py
cp decoders/templates/__init__.py decoders/challengeDecoders/

(two underscores on each side)

The second command creates a directory, and the third copies a template for TCP traffic to a new Python file we will work in (only use a single space between the two file paths shown above on separate lines).
The fourth command copies a blank file that is needed since we are writing Python code (Dshell uses this to identify that our folder contains a decoder).

****

Dshell contains templates for two types of decoders. One looks at packets individually (PacketDecoder.py) and the other (SessionDecoder.py) looks at reassembled streams of data that make up a sort of ***dialog*** of back-and-forth communication. For this exercise, since you want reassembled TCP traffic, you have made a copy of SessionDecoder.py that you will customize.

****

9) Now let us modify the decoder template. Open the file by running the following command to use the text editor ***Geany*** (***vim*** is also available on this machine).

geany decoders/challengeDecoders/archdeusSession.py &

****

You will notice several functions in the SessionDecoder.py template file. These allow you to specify logic that will be called at different times, as described in the following (comments are also included in the source code).

```
def __init__(self, **kwargs) – Called during instantiation of the decoder. Specify BPF filter, name, author, and description of decoder here.
def packetHandler(self, udp, data) – Called when a UDP packet is received. Specify parse logic here. To handle individual TCP packets, use the IPDecoder class (see the PacketDecoder.py template as an example).
def connectionInitHandler(self, conn) – This is called when the connection is closed, either the connection ends or it reaches the maximum size allowed my Dshell.
def blobHandler(self, conn, blob) – One or more packets sent unidirectionally make up a blob. This is called after a data stream changes direction from client to server or server to client; the data of all of the packets for that data stream session are reassembled.
def connectionHandler(self, conn) – Called after all packets that make up a session have been reassembled.
def connectionCloseHandler – Called when the connection between communicating entities ceases.
```

****

10) Start by giving your dissector the name we chose earlier (archdeusSession) that will be used by Dshell. If you want to change the name, be sure to also update the name of the file previously created (decoders/challengeDecoders/archdeusSession.py). Change the constructor (that is, the `__init__` function) to reflect what you see in Fig. 4 (choose your own values for name [if updated], description, and author). Include a ***filter*** using a BPF filter to ensure you only work with TCP data, port 9999 (look back at question 7)a if you completed the first software reverse-engineering exercise). ***(Note: code indention matters.)***

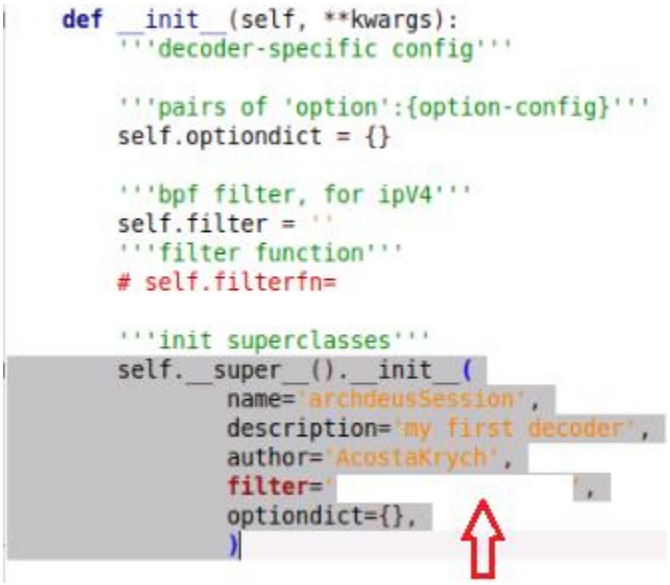

Fig. 4 __init__function in Dshell's SessionDecoder.py template file

11) Save your file by Clicking on **File** -> **Save**.

12) Go back to the terminal and ensure that your new decoder is picked up by Dshell. *(Note: If you are no longer within the Dshell shell, drop back into it by running* **./dshell** *in the terminal.) (Hint: scroll up; it should be at the top; it is easiest to look for it based on the author column.)*

decode -l

13) Add the following to your code, shown in the gray boxes in Fig. 5. This will print the time when a TCP "conversation or dialog" is encountered. Recall that code indention matters.

Fig. 5 Dshell's SessionDecoder.py template file

14) Save your file and test your decoder by running the following in your terminal. *(Note: you should see at least one line of output; otherwise, your capture may be too short.)*

decode -d archdeusSession mycap.pcapng

15) Now let us print the data for all of the packets that belong to the malware communication. Change the `blobHandler` to reflect the following in Fig. 6. The decoder will now print the data (called **blob** here) from the packets that match our BPF filter. Recall that code indention matters.

```
def blobHandler(self, conn, blob):
    '''handle session data as soon as reassembly is possible'''
    encoded_message = str(blob)
    print('Packet data found: ' + encoded_message)
```

Fig. 6 blobHandler function in Dshell's SessionDecoder.py template file

16) Save your file and test your decoder by running the following in your terminal. *(Note: use your new decoder name if you changed it.)*

    decode -d archdeusSession mycap.pcapng
    (You should see a few lines of output; otherwise, your capture could be too short.)

17) Now we are ready to decode the data. Modify the `blobHandler` function again. This time, you need to implement the decoding algorithm for ROT-13. Recall that this was the encoding algorithm the RAT was using to obfuscate its C2, which you identified in the previous exercise. (*Hint: make sure to convert the blob to a string before you work with it using Python casting, [i.e.,* ***str(blob)*** *as shown in Fig. 5.])*

    Additionally, you may choose to adopt the following Python ROT-13 code shown in Fig. 7 into your decoder.

rot13.py

```
1  encoded_message = "hello" #change this to instead read str(blob)
2  encoded_message = encoded_message.lower() #conver to lower case
3  decoded_message = "" #will hold output
4
5  #work on one character at a time
6  for character in encoded_message:
7      #get the numeric (ASCII) value of the character
8      character_number = ord(character)
9      #increase or decrease by 13; depending on the character value
10     if 'a' <= character <= 'm':
11         character_number = character_number + 13
12     if 'n' <= character <= 'z':
13         character_number = character_number - 13
14     #change the value back into an (ASCII) character
15     new_character = chr(character_number)
16     #add the encoded character to our decoded message
17     decoded_message = decoded_message + new_character
18 #after the for loop is complete, we print the decoded value message
19 print(decoded_message)
```

Fig. 7 ROT-13 Python code

18) Remember to save and test your decoder often. Once you are able to decode all of the messages, describe the malware's behavior in your own words here:

Great job! You have successfully analyzed a binary and you can now decode all of the traffic between the C2 server and Bot!

If you are out of time or stuck, you can copy over the solution with the following command:

sudo cp /root/sln/archdeusSession.py /home/student/Dshell/decoders/challengeDecoders/

6. Conclusion

After completing this exercise, participants should have a better understanding of how RATs work, how rotation ciphers work, what ARL's network forensic analysis framework Dshell is and is capable of, and how to develop and use a Dshell decoder to uncover and decode malware communications. This exercise and the related reverse-engineering exercises have been, and will be, shared with collaborators and partners (including professionals, faculty, and students) to help establish a common ground for studying, researching, and learning about malware, analysis tools, and analysis techniques, to harden systems and to develop ways to recover after compromise and detect and protect systems moving forward.

We have received positive feedback on these software reverse-engineering exercises from both cybersecurity novices and experts.

Future exercises could entail a broader scope of Dshell and how to fully leverage the framework, such as by conducting advanced network forensics and analysis on a network for threat hunting, learning how to modify and customize existing decoders to fit an end-user's analysis needs, learning how an existing decoder was developed to decode a specific network protocol, or developing a decoder to detect and decode real, complex network protocols found in the wild today. Additionally, ARL recently developed a new version of Dshell written in Python 3, which they plan to release publicly, and could be used in future exercises.

We hope the information found herein will enlighten students, researchers, and practitioners in the cybersecurity field with new analysis tools and techniques, and help spawn ideas to better their security posture and develop new and unique Dshell decoders.

7. References

1. Acosta JC, Krych DE. Hands-on cybersecurity studies: uncovering and decoding malware communications—initial analysis with Wireshark and Volatility. Adelphi (MD): CCDC Army Research Laboratory (US); Forthcoming 2020.
2. Acosta JC, Krych DE. Hands-on cybersecurity studies: uncovering and decoding malware communications—malware analysis with Ghidra. Adelphi (MD): CCDC Army Research Laboratory (US); Forthcoming 2020.
3. Wireshark. [accessed 2020 March]. https://www.wireshark.org.
4. Volatility. [accessed 2020 March]. https://www.volatilityfoundation.org.
5. Ghidra. [accessed 2020 March]. https://ghidra-sre.org.
6. Dshell. [accessed 2020 March]. https://github.com/USArmyResearchLab/Dshell.
7. VirtualBox. [accessed 2020 March]. https://www.virtualbox.org/.
8. Microsoft Windows 7. [accessed 2020 March]. https://www.microsoft.com/en-us/software-download/windows7.
9. Ubuntu 18.04. [accessed 2020 March]. https://releases.ubuntu.com/18.04.4/.

List of Symbols, Abbreviations, and Acronyms

ARL	Army Research Laboratory
BPF	Berkeley Packet Filter
C2	command and control
CCDC	US Army Combat Capabilities Development Command
CyberRIG	Cyber Rapid Innovation Group
DHCP	dynamic host configuration protocol
IP	Internet Protocol
pcap	packet capture
RAT	remote access trojan
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	virtual machine

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	CCDC ARL FCDD RLD CL TECH LIB
2 (PDF)	CCDC ARL FCDD RLC ND D KRYCH J ACOSTA