DNS-Based Command and Control (C2) Server for Firewall and Antivirus Evasion

Jeffrey DiVincent
University of Central Florida
Orlando, USA
divi@knights.ucf.edu

Christopher Fischer
University of Central Florida
Orlando, USA
fischer@knights.ucf.edu

Matthew McKeever
University of Central Florida
Orlando, USA
mmckeever@knights.ucf.edu

ABSTRACT

Advanced and sophisticated attackers, especially those funded by nation-states, will use uncommon techniques to obfuscate their activity on a network. This includes using esoteric communication channels instead of more traditional channels like HTTP(S) to communicate with a command-and-control (C2) server for maintaining persistence and executing post-exploitation tasks. However, these atypical channels are seldom used in penetration tests, which leaves gaps in an organization's security posture that advanced attackers can exploit. We will be developing a Python3-based command-and-control agent and server and testing its effectiveness against compromising residential networks, regardless of client operating system and network speeds. To benefit the wider cybersecurity industry, we will be open sourcing our code under a libre software license to make it available to individual penetration testers and smaller organizations.

KEYWORDS

command and control; malware; firewall; antivirus; antimalware cybersecurity; python; dns

ACM Reference Format:
Jeffrey DiVincent, Christopher Fischer, and Matthew McKeever. 2022. DNS-Based Command and Control (C2) Server for Firewall and Antivirus Evasion. In *Foundations of Computer Security and Privacy*. ACM, New York, NY, USA, 7 pages. https://doi.org/not_applicable

1 INTRODUCTION

Advanced attackers, such as those financed by nation-states (a.k.a. active persistent threats, or APTs), are known to use novel, relatively unknown techniques to communicate back to a command-and-control (C2) server they control to maintain persistence and avoid detection. However, a lot of tooling that red-team security researchers utilize do not utilize these novel communication techniques, opting for traditional C2 communication channels such as HTTP(S). This is especially true for smaller companies and governments who cannot afford premium solutions like Fortra's *Cobalt Strike*, which offers harder-to-detect DNS command-and-control capabilities at a hefty premium. By only attacking with C2s that use traditional communication channels, a gap in coverage is created, leaving companies, governments, and individuals unknowingly vulnerable to data exfiltration and more sophisticated attacks. One of these novel data exfiltration techniques involve utilizing DNS queries to communicate between an attacker's server and an infected host. We wish to develop a DNS-based command and control server and client that can be freely used by security professionals, and then see if it is harder to automatically detect with residential-grade antimalware products like Windows Defender compared to traditional HTTP C2. We will also ensure that our solution is robust on a variety of operating systems and networking conditions. We desire to release our proof-of-concept C2 server as a free-to-use, permissively licensed open-source project for individual security researchers and smaller organizations to utilize.

Organization. The organization of this paper is as follows. In section 3, we review the existing ecosystem of DNS-based command-and-control servers. In section 4, we discuss the approach we took for our C2 server and how the agent communicates with it. This includes discussing the design of the protocol. In section 5, we talk about licensing and distribution of our work, and our rationale for releasing our proof of concept under a libre software license. Section 6 describes our approach to test how covert our approach is, and Section 7 describes how we quantified network performance. We then report our findings in section 8, analyze them in section 9, present our work distribution in section 10, and our final remarks in section 11.

2 RATIONALE

Our rationale behind this research project was to develop a C2 that can be used in collegiate cybersecurity competitions such as Collegiate Cyber Defense Competition (CCDC), Information Security Talent Search (ISTS), and NSA's Center of Academic Excellence National Cyber Exercise (CAE-NCX), along with Attack & Defend and King of The Hill (KoTH) -styled competitions, much like other schools such as the Rochester Institute of Technology (RIT). We also pursued this project to gain a deeper understanding of the process and methodologies to developing a custom C2 framework over an arbitrary protocol, and we chose DNS specifically due to its ubiquity and the high likelihood that it is not blocked on a given network.

3 RELATED WORK

While this technique is novel, it is not completely new in the Command and Control world. Past literature shows that both nation state and commercial criminal entities have deployed botnets which use DNS as a carrier for command and control. Feederbot, a commercial botnet, was known to leverage DNS Command and Control

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CAP5150, December 5, 2022, Virtual Event
© 2022 Association for Computing Machinery.
ACM ISBN 978-0-0000-0000-0/00/00…$0.00
https://doi.org/not_applicable

[2]. Similarly, APT-18 utilized DNS Command and Control in an attack against a Commercial Entity in the United States, according to Palo Alto Research [3]. APT-18 is commonly attributed to the Chinese Government's Offensive Cyber Capability.

DNS is also used as a command-and-control channel in both commercial and open-source tools marketed towards security researchers. The open-sourced command and control suite *Sliver* implements C2 over DNS as one of its built-in agents [1]. How Sliver designed its C2 is that its server is listed as a name-server of an attacker-controlled domain. While this does allow for normal DNS servers (including a company's internal DNS servers) to be called to communicate to the C2 server, it requires log entries that all point towards a single attacker-controlled domain. This causes the attacker's domain to be a clear indicator of compromise (IoC), allowing for easy blocking. While this is similar to tactics used by adversaries, it discounts other approaches that may be taken, including communication to a custom DNS server directly.

A more robust command and control suite is Fortra's *Cobalt Strike*. While Cobalt Strike does support Sliver's approach to DNS, it also is heavily customizable, also including the ability to listen on multiple domains [5]. Using a large roster of domains, which can be obtained from free-domain registrars like *Freenom* (which offers .TK, .GA, and other top-level domain [TLD] names for free) or using cheap TLDs like .XYZ, this can make it harder to be simply filtered out. However, while Cobalt Strike is harder to detect, it is also expensive; licenses start at $9,450 per user per year [4]. This prices out most benign users, but does not prevent determined cyber-criminals from purchasing a license, assuming they do not just pirate it.

While our approach will be able to be used as a name-server similar to other approaches, it will be designed to operate as a standalone DNS server, which also grants us great powers over how we send and receive packets.

4 APPROACH

We have designed a DNS-based protocol for communication between a "victim" computer and a server. By designing a concrete protocol, we can ensure that the client and the server, no matter what language either are written in, operate consistently. Data exchange is done as a domain name query/answer format; the victim periodically will query the server if an action is to be done, and then the response will determine whether a command should be run. To ensure persistence, an agent will be deployed onto the victim host that will run in the background on log-in.

Because DNS traffic is almost never blocked, especially on residential networks where deep-packet inspection is not being done to check for malicious traffic, this allows for command-and-control data transmission to be allowed through the firewall where it may otherwise be detected. While it may be possible to detect non-obfuscated DNS C2 traffic with packet inspection using high-end anti-malware solutions like VMWare's *Carbon Black* or CrowdStrike's *Falcon* platform, we will not focus our study on this use case due to a cost barrier to acquire these business-oriented products. Despite this, residential targets are still valuable for attackers, as they can be used to create a botnet to later launch denial-of-service attacks.

Figure 1: The data exchange format of DNS answers sent from the C2 server.

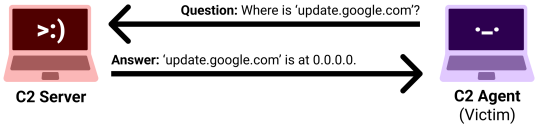


Figure 2: Data exchange when no commands are being run.

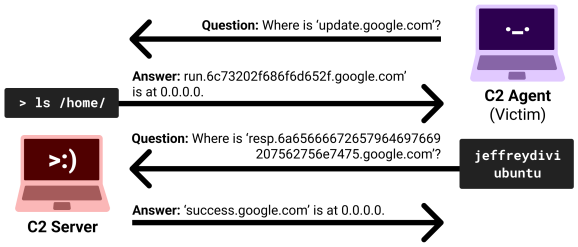


Figure 3: Data exchange when a command is being executed.

It is also harder to manually detect C2 DNS traffic; while we will not yet be obfuscating C2 traffic due to this being a proof-of-concept, the commonality of DNS traffic makes it harder to spot C2 traffic if analyzed by a dedicated security operations team.

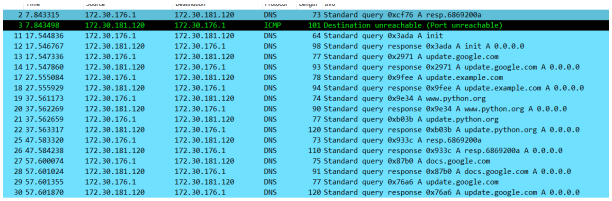
Protocol. The protocol is designed to be simple to implement. Commands are structured as domain queries and answers in the structure of *<command>.<data>.<anything else>*. The command section is a valid command to send to the server, while the data section is a base16-encoded string using a lowercase alphabet. The following are valid commands that can be sent to the server:

- Sending the "update" command to the server will fetch if a new command has been issued from the server operators.
- Sending the "resp" command to the server will send requested data to the C2 server.

The following are valid responses sent from the server:

- "success" and "failure" are status codes for valid commands.
- "run" instructs the agent to run a command in a shell, and then make a "resp" query with the response to the server.
- Everything else will be treated as a no-op. This is also used for obfuscation of periodic traffic, including disguising our traffic as normal web traffic to popular websites like Google, Amazon, and Microsoft.

When *idle*, the protocol acts somewhat normal. The agent periodically sends obfuscated "update" commands to the server, and the server responds with a no-op back. This is depicted in Figure 2. However, when a command is requested by the server (i.e. the network is *not idle*), the next time the "update" command is requested by the agent, it will receive a command. This command is then run and the response is sent to the server. When the server receives the agent's response, it returns an affirmative, and the agent is set to *idle*. This is depicted in Figure 3.



2 7.843315	172.30.176.1	172.30.181.120	DNS	73	Standard query 0xcf76 A resp.6869200a
3 7.843498	172.30.181.120	172.30.176.1	ICMP	101	Destination unreachable (Port unreachable)
11 17.544836	172.30.176.1	172.30.181.120	DNS	64	Standard query 0x3ada A init
12 17.546767	172.30.181.120	172.30.176.1	DNS	98	Standard query response 0x3ada A init A 0.0.0.0
13 17.547336	172.30.176.1	172.30.181.120	DNS	77	Standard query 0x2971 A update.google.com
14 17.547860	172.30.181.120	172.30.176.1	DNS	93	Standard query response 0x2971 A update.google.com A 0.0.0.0
17 27.555084	172.30.176.1	172.30.181.120	DNS	78	Standard query 0x9fee A update.example.com
18 27.555929	172.30.181.120	172.30.176.1	DNS	94	Standard query response 0x9fee A update.example.com A 0.0.0.0
19 37.561173	172.30.176.1	172.30.181.120	DNS	74	Standard query 0x9e34 A www.python.org
20 37.562269	172.30.181.120	172.30.176.1	DNS	90	Standard query response 0x9e34 A www.python.org A 0.0.0.0
21 37.562659	172.30.176.1	172.30.181.120	DNS	77	Standard query 0xb03b A update.python.org
22 37.563317	172.30.181.120	172.30.176.1	DNS	120	Standard query response 0xb03b A update.python.org A 0.0.0.0
25 47.583320	172.30.176.1	172.30.181.120	DNS	73	Standard query 0x933c A resp.6869200a
26 47.584238	172.30.181.120	172.30.176.1	DNS	110	Standard query response 0x933c A resp.6869200a A 0.0.0.0
27 57.600074	172.30.176.1	172.30.181.120	DNS	75	Standard query 0x87b0 A docs.google.com
28 57.601024	172.30.181.120	172.30.176.1	DNS	91	Standard query response 0x87b0 A docs.google.com A 0.0.0.0
29 57.601355	172.30.176.1	172.30.181.120	DNS	77	Standard query 0x76a6 A update.google.com
30 57.601870	172.30.181.120	172.30.176.1	DNS	120	Standard query response 0x76a6 A update.google.com A 0.0.0.0

Figure 4: Wireshark traffic between the command-and-control server and the agent (client).

While the simplicity of the protocol makes it easier to decode, since our primary goals are to make a stable proof-of-concept and to test it with residential antimalware products, this is not of concern. Despite this, we will be applying some basic obfuscation by disguising traffic as subdomains of random domains.

While we did adopt base16 to store commands for this proof of concept, we could reasonably adopt a modified base32 encoding, or even a base64 encoding scheme; while DNS is a case-insensitive protocol, during development we discovered that we can send and receive with casing preserved. We deemed abusing this finding in our C2 infrastructure would be seen as suspicious and did not implement it.

Server. The command-and-control server is a Python3 application using the dnslib library. Python3 itself provides memory safety while having a syntax that allows for fast prototyping, but this comes at the cost of performance. For a server, though, this is a worthy trade-off. dnslib is an open-source PyPi library that allows for DNS traffic manipulation, including parsing incoming DNS requests and crafting the outgoing response packets. It also has embedded capabilities to be used as a lightweight and customizable DNS server, allowing us to easily implement our designed query protocol in code. At the moment, the server has the ability to receive and reply to DNS traffic with a basic command language implemented; it is a functional proof-of-concept command-and-control server. Scapy, a general packet manipulation library, was also considered, but was not chosen due to dnslib's server capabilities.

Client. The command-and-control agent is a Python3 application using the *dnslib*, *subprocess*, and *socket* libraries. Currently, the agent can perform the following actions: callback to server, poll server periodically for commands to execute, periodically queries legitimate domains, and data exfiltration. The agent uses dnslib's *DNSRecord* object to send DNS queries to the server and parses the answer packet to read the instructions sent by the C2 server and uses *subprocess* to execute commands on the infected host.

The initial design utilized Python 3's struct and socket modules to construct DNS query packets manually, but this was phased out in favor of *dnslib*'s capabilities. Additionally, Python3 was chosen over C or C++ due to certain Windows API calls being flagged as suspicious by AV and Windows Defender and prompting a scan of the binary. Moreover, the module support in Python3 makes implementing and improving functionality more manageable in our proof-of-concept.

```python
# Build a response query
def build(req, body):
    reply = req.reply()
    reply.add_answer(*RR.fromZone(body + ". 60 A 0.0.0.0"))
    return reply

# Process a question sent by agents
def execute(req, ip):
    domain = str(req.get_q().get_qname())[:-1]
    data = domain.split(".")

    # init.
    if data[0] == "init":
        client_id = add_user(ip)
        return build(req, "success." + random_nop())

    # resp.<data>.
    elif data[0] == "resp":
        # print(data[1])
        hex_code = base64.b16decode(data[1].upper()).decode("utf8")
        print(f"<{ip}> {hex_code}")
        return build(req, "success." + random_nop())

    # update.
    elif data[0] == "update":
        if command == "":
            return build(req, domain)
        else:
            # run.<command>.
            return build(req, "run." + base64.b16encode(command.encode("utf8")).decode("u

    # NOP
    else:
        firstChar = str(domain)[0]
        if firstChar == "0" or firstChar == "1" or firstChar == "2" or firstChar == "3" o
            return build(req, random_nop(subdomain=True))

        return build(req, domain)
```

Figure 5: A Python3 code snippet from the command-and-control server showing the logic that implements the command transmission protocol.

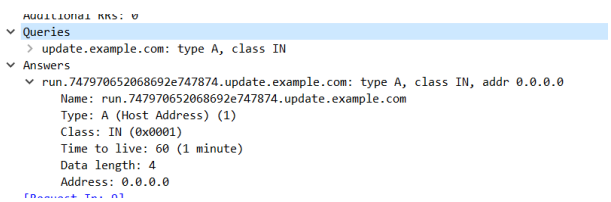


Figure 6: A DNS answer sent by the agent to the command-and-control, displayed in Wireshark.

Traffic Obfuscation. For our proof of concept, we disguised all C2 traffic as legitimate-looking traffic to popular domains, including *google.com*, *microsoft.com*, *stackoverflow.com*, and *slack.com*. Because we control the DNS server and it is exclusively used for command and control, we can 'lie' about the responses to disguise our traffic; domains do not need to be valid to be sent. Command names, stored in the leftmost subdomain, are not obfuscated at all in our proof of concept, which can be improved in the future by mapping commands with normal-looking subdomains like "*www*," "*mail*," and "*help*." Additionally, the C2 agent periodically queries legitimate-looking domains to our C2 server to "drown" our malicious traffic with seemingly normal DNS queries and responses (see Figure 4).

Testing. We performed two types of tests on our C2 network. First, we ran network degradation tests on our developed agent and its server. This let us quantify the reliability of DNS as a command and control protocol on different network conditions and lets us test for potential faults like packet loss and network congestion. We elaborate on our procedure for this in section 7. We also tested our C2 network against common non-cloud-based intrusion prevention systems, including SNORT IDS on Linux and a combination of Windows Defender and Windows Firewall on Windows. We elaborate on our procedure for this in section 6. These two tests let

us quantify both the reliability of our solution and how covert it is on a network.

5 SOFTWARE DISTRIBUTION

A goal of this project is to release our source code and findings openly to security researchers and penetration testers to better secure networks. We have publicly released source code on GitHub under the MIT license, as it is a libre software license that gives teams of all sizes the ability to benefit from our research without having to worry about license compliance. However, this also comes at the cost of malicious actors utilizing our findings as well. To mitigate this, our agent will also be released as a Python script instead of an executable, marginally heightening the barrier of entry to utilize the software to deter illicit use. Python is an interpreted language, so traditional distribution techniques require the source code be stored on disk when run, allowing for easy detection if used maliciously.

Additionally, the released C2 agent and server, while they can be used for penetration tests and internal red-team engagements, are not practical for mass data exfiltration. the C2 only supports running, and returning the output of, commands. While exfiltration of files is possible (i.e. by using *cat* and Unix shell piping) it is limited by the maximum length of (sub)domains in DNS, the base16 encoding, and the intentional lack of any measure to circumvent these shortcomings. This is fine for controlled offensive security engagements, but impractical for real-world exfiltration.

Overall, we believe that the potential benefit of releasing our code to the cybersecurity field outweighs the potential costs.

As all of our researchers are also members of the University of Central Florida's Collegiate Cybersecurity Competition (UCF C3) team, we will also be trialing and expanding upon this tool in offensive-security cybersecurity competitions and as a training tool for defensive-security competitions, allowing the next generation of cybersecurity professionals to be aware of how to identify esoteric command-and-control channels.

6 INTRUSION PREVENTION TESTING

Testing Network and Host Based Intrusion systems is paramount to ensuring our technique works. This testing will be performed on a variety of systems. It should be noted that we are not considering endpoint detection and response systems with significant cloud-based offerings in scope for our design and testing. Linux network intrusion detection and prevention testing will be accomplished by utilizing off-the-shelf antivirus tooling for disk-based signatures coupled with SNORT IDS running in the network for network-based signature detection. Windows testing involves running our sample on multiple versions of the Windows Server and Client based operating systems. These hosts will have all features of Windows Defender enabled, except for such features that will upload our implant to the cloud.

7 NETWORK DEGRADATION TESTING

Our network degradation testing technique relies on testing two variables simultaneously. First, our testing strategy incorporates the design of multiple Docker images with different operating system baselines. This includes Linux distributions of Red Hat Linux (CentOS), Debian, Ubuntu, and Alpine Linux. For our selection of versions, we choose the two most recently released Long Term Supported (LTS) versions. On distributions where no delineation is made between releases for designations of support period, we simply use the two most recent releases. Our reasoning is that we want our implant to be supported on the most recent version of the operating system, as opposed to historical releases which will include larger technical debt to enable similar functions. By testing across operating system types and releases, we force ourselves to follow a programming convention where our code is portable to different operating system versions and releases (and thus widely supported).

The second variable which we will be testing is network conditions (broadly). We will be testing differences. Broadly, we will have three categories of network profiles. We will account for total bandwidth (10mbps, 5mbps, and 1mbps), total speed (100ms latency, 500ms latency, and 1500ms (about 1 and a half seconds) latency), and total packet loss (0%, 15%, and 30%). Each network profile will be tested with the other profiles, leading to a total of 27 tests per operating system version. Our network conditions will be applied via the Linux utility *tc*. These changes will be implemented on the same Docker image via an agent which coordinates with our testing framework.

The test case will be a simulated command sent to the host. The server will always be run on a consistent machine (in this case, an Ubuntu 22 LTS release). The server will queue for the command to be sent to a virtualized host. The agent on the machine which applies network conditions will also be looking for this command to be run. In our case, we will be adding a test command to create an IPC call via the Linux kernel to alert our agent in the case of a successful test. If no call is received 60 seconds after the testing has started, the test is marked as a failure.

Each virtual machine and network profile will be run in parallel 10 times. The accuracy of the implant will be determined by a total percentage (pass/fail). Our expectation is that, given DNS being a lossy service (due to it being done over UDP), some tests will fail due to bad timing. Our agent should be able to detect when no result is received from the server, or when a connection is not fully established; in this case, it should re-transmit the command and execute the command before the time expires. However, the presence of an unusually high failure rate may be indicative of a logic error or programming issues related to networking.

8 RESULTS

The technique when applied shows promise in making covert communication harder to analyze. When tested on a network with a SNORT IDS, our DNS command-and-control traffic did not trigger any specific alerts. Additionally, when tested on an Windows 10 system with a fully enabled Windows Defender, neither the agent or server was flagged as malicious or quarantined while at rest or during execution. Furthermore, the Windows Firewall did not drop any of the inbound or outbound DNS packets from our server and agent. Moreover, viewing the network traffic on WireShark (see Figure 4) illustrates that the DNS traffic by the C2 agent and server looks mostly benign and that further packet analysis is required to locate what commands are being transmitted (see Figure 6).

Table 1: Network degradation testing: Speed capped at 10 Mbps

Operating System	Speed	Passed Tests	Reliability
CentOS Linux 7.0-1406	10mbps	10	100%
CentOS Linux 8.0-1905	10mbps	10	100%
Debian 10	10mbps	10	100%
Debian 11	10mbps	10	100%
Ubuntu 20.04	10mbps	10	100%
Ubuntu 22.04	10mbps	10	100%
Alpine 3.16	10mbps	10	100%
Alpine 3.17	10mbps	10	100%

Table 2: Network degradation testing: Speed capped at 5 Mbps

Operating System	Speed	Passed Tests	Reliability
CentOS Linux 7.0-1406	5mbps	10	100%
CentOS Linux 8.0-1905	5mbps	10	100%
Debian 10	5mbps	10	100%
Debian 11	5mbps	10	100%
Ubuntu 20.04	5mbps	10	100%
Ubuntu 22.04	5mbps	10	100%
Alpine 3.16	5mbps	10	100%
Alpine 3.17	5mbps	10	100%

Table 3: Network degradation testing: Speed capped at 1 Mbps

Operating System	Speed	Passed Tests	Reliability
CentOS Linux 7.0-1406	1mbps	10	100%
CentOS Linux 8.0-1905	1mbps	10	100%
Debian 10	1mbps	10	100%
Debian 11	1mbps	10	100%
Ubuntu 20.04	1mbps	10	100%
Ubuntu 22.04	1mbps	10	100%
Alpine 3.16	1mbps	10	100%
Alpine 3.17	1mbps	10	100%

Table 4: Network degradation testing: Latency set to 100ms

Operating System	Latency	Passed Tests	Reliability
CentOS Linux 7.0-1406	100ms	10	100%
CentOS Linux 8.0-1905	100ms	10	100%
Debian 10	100ms	10	100%
Debian 11	100ms	10	100%
Ubuntu 20.04	100ms	10	100%
Ubuntu 22.04	100ms	10	100%
Alpine 3.16	100ms	10	100%
Alpine 3.17	100ms	10	100%

Network Degradation Testing - Speed. We did not see any degradation or failure from changing speed as listed. This is likely because the speed was sufficient as to not cause latency for our traffic and not result in any drops. Data is shown in Tables 1, 2, and 3 for speeds of 10 Mbps, 5 Mbps, and 1 Mbps respectively.

Network Degradation Testing - Latency. We did not see any degradation or failures from changing latency as listed. Our C2 infrastructure had a 100% command execution attempt-to-success rate regardless of latency. Data is shown in Tables 4, 5, and 6.

Table 5: Network degradation testing: Latency set to 500ms

Operating System	Latency	Passed Tests	Reliability
CentOS Linux 7.0-1406	500ms	10	100%
CentOS Linux 8.0-1905	500ms	10	100%
Debian 10	500ms	10	100%
Debian 11	500ms	10	100%
Ubuntu 20.04	500ms	10	100%
Ubuntu 22.04	500ms	10	100%
Alpine 3.16	500ms	10	100%
Alpine 3.17	500ms	10	100%

Table 6: Network degradation testing: Latency set to 1500ms

Operating System	Latency	Passed Tests	Reliability
CentOS Linux 7.0-1406	1500ms	10	100%
CentOS Linux 8.0-1905	1500ms	10	100%
Debian 10	1500ms	10	100%
Debian 11	1500ms	10	100%
Ubuntu 20.04	1500ms	10	100%
Ubuntu 22.04	1500ms	10	100%
Alpine 3.16	1500ms	10	100%
Alpine 3.17	1500ms	10	100%

Table 7: Network degradation testing: 0% packet loss

Operating System	Packet Loss	Passed Tests	Reliability
CentOS Linux 7.0-1406	0%	10	100%
CentOS Linux 8.0-1905	0%	10	100%
Debian 10	0%	10	100%
Debian 11	0%	10	100%
Ubuntu 20.04	0%	10	100%
Ubuntu 22.04	0%	10	100%
Alpine 3.16	0%	10	100%
Alpine 3.17	0%	10	100%

Table 8: Network degradation testing: 15% packet loss

Operating System	Packet Loss	Passed Tests	Reliability
CentOS Linux 7.0-1406	15%	9	90%
CentOS Linux 8.0-1905	15%	7	70%
Debian 10	15%	9	90%
Debian 11	15%	7	70%
Ubuntu 20.04	15%	9	90%
Ubuntu 22.04	15%	8	80%
Alpine 3.16	15%	10	100%
Alpine 3.17	15%	8	80%

Network Degradation Testing - Packet Loss. Packet loss approximately maps to a failing test; the more packets lost, the more our C2 infrastructure fails. This is not surprising, as UDP-transmitted data will be lost if the packet is dropped during transit. That is, if the packet that sends the command is drop, the test will fail. Data is shown in Tables 7, 8, and 9.

9 DISCUSSION

Limitations. At present, the command and file transfer length is greatly limited by the DNS protocol and our encoding method; the size of DNS packets and maximum length for domain names, as described in the RFC, restricts the amount of data that can be

Table 9: Network degradation testing: 30% packet loss

Operating System	Packet Loss	Passed Tests	Reliability
CentOS Linux 7.0-1406	30%	9	90%
CentOS Linux 8.0-1905	30%	8	80%
Debian 10	30%	8	80%
Debian 11	30%	7	70%
Ubuntu 20.04	30%	8	80%
Ubuntu 22.04	30%	5	50%
Alpine 3.16	30%	9	90%
Alpine 3.17	30%	5	50%

exfiltrated and length of commands to be executed in a single packet [6]. Additionally, the UDP protocol is unreliable as it can drop or lose packets during transmission, which reduces the effectiveness of data exfiltration. Furthermore, detection of traffic generated by the C2 framework is trivial for humans due to the encoding method used in our framework.

Future Improvements. To further improve upon this C2 framework, the first thing we can do is create a more robust, and modular, command set. Currently, the server is only capable of sending commands to all connected agents and collect their responses. While this is immensely useful for persistence, it does not have the proper ability to exfiltrate files or perform common post-exploitation actions such as privilege escalation or credential harvesting. While the designed protocol allows for this, by reworking the command parser to be more modular, we can make common tasks trivial, which is especially useful for offensive security professionals performing engagements.

Similarly, our C2 also does not have a proper user interface. All commands are sent by a short-lived terminal-like UI that is not run as an always-on daemon. By both splitting the server into an always-running daemon and providing a more user-friendly UI, our server can be used in more permanent installations, which is a feature many other C2 solutions (including Sliver) already include. This also can provide for improved management of multiple victim computers, including the ability to self-destruct and clean itself up after engagements.

Additionally, our framework should implement splitting data and commands across multiple DNS queries and responses to circumvent the issue described previously in this section. Our current design requires commands or data to be sent over a single packet; this improvement will enable execution of longer commands and exfiltration of larger files like a database dump or password hashes. Furthermore, devising a better method for packet obfuscation would improve our evasion and detection efforts for our malicious C2 traffic. The current obfuscation method can be trivially detectable by most humans because of our base16 encoding in subdomains, so to be used as a tool in offensive security, a better obfuscation method is required.

Moreover, adding more robust error handling for both Python3 and the UDP protocol would improve efficiency and stability in our framework. For Python3, an example would be better exception handling because a malformed packet could cause undefined behavior or crash the server/agent entirely. For UDP, an example would be implementing either reliable data transfer using UDP or devising a custom packet redundancy strategy to minimize packet loss.

10 WORK DISTRIBUTION

Jeffrey DiVincent served as the team lead and has developed a basic DNS command-and-control server and data transfer protocol to be used by the client. Developing the C2 server took approximately five hours, with the data protocol being developed simultaneously based on implementation. DNS testing was done using Windows's *nslookup* utility. He also handled the structure of the milestone document and worked with Matthew McKeever on research for how to implement the agent, which took two hours. Project management, in general, took approximately three additional hours when summed together.

Christopher Fischer served as a technical researcher and advisor. He has coordinated how to best apply the technique and obfuscation method for the traffic itself based on both experience, tests performed in a lab environment, and research based on past malware samples known to contain DNS Command and Control components.

Matthew McKeever served as C2 agent developer and collaborated with Jeffrey DiVincent on server development. He also has coordinated with Jeffrey on the implementation of DNS log suppression and domain and traffic obfuscation in the server and agent. Agent testing was performed with the server to verify functionality of the initialization/callback, polling, command execution, and data exfiltration. The development of the agent PoC took approximately four hours and the additional server features took two hours.

Since the proposal, we unfortunately had team-member Nathan Nau drop the class, so he is no longer involved in this project.

11 CONCLUSION

We developed a novel, custom C2 framework that communicates over the DNS protocol using domain queries and A-record responses. Our C2 server poses as a fake, legitimate DNS resolver and sends command as an A-record response to the agent's update query. Additionally, we found that our C2 could successfully evaded detection by residential-grade antimalware products like Microsoft Windows Defender and Intrusion Detection/Prevention Systems (IDS/IPS) like Snort. We also found that our C2 communication traffic had negligible impact on overall network performance and minimal packet loss without implementing any form of reliable data transfer.

Next Steps. To further our project, the next steps would be to research other protocols and techniques that could be used for persistent command-and-control servers and testing in a more 'realistic' environment. For example, some protocols and techniques would be MQTT - a messaging protocol used by IoT devices, DNSSEC, kernel modules (Linux) and drivers (Windows), process injection, and polymorphic or metamorphic malware. Additionally, testing in collegiate cybersecurity competitions would provide a safe and realistic environment to evaluate our C2's effectiveness in evading detection against blue-teamers and advanced Intrusion Detection Systems (IDS) and stateful firewalls.

Acknowledgement. This work was written for CAP 5150 at the University of Central Florida and will be adopted by the University of Central Florida Collegiate Cybersecurity Competition (C3) Team.

REFERENCES

[1] Joe . 2019. DNS C2 · BishopFox/sliver Wiki. https://github.com/BishopFox/sliver/wiki/DNS-C2

[2] 2011 Seventh European Conference on Computer Network Defense 2011. *On Botnets That Use DNS for Command and Control.* 2011 Seventh European Conference on Computer Network Defense. https://doi.org/10.1109/ec2nd.2011.16

[3] Josh Grunzweig. 2016. New Wekby Attacks Use DNS Requests as Command and Control Mechanism. https://unit42.paloaltonetworks.com/unit42-new-wekby-attacks-use-dns-requests-as-command-and-control-mechanism/

[4] Fortra LLC. [n.d.]. Cobalt Strike Pricing - Cobalt Strike Research and Development. https://www.cobaltstrike.com/pricing-plans/

[5] Fortra LLC. [n.d.]. DNS Beacon. https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/listener-infrastructue_beacon-dns.htm

[6] P. V. Mockapetris. 1987. RFC1035: Domain names - Implementation and Specification. Internet Requests for Comments. https://doi.org/10.17487/RFC1035