Reverse Engineering CISCO OpenConnect (October 2019)

Arvind P Jayan, Sai Kiran Uppu, Debolina De

Abstract— The task was to reverse engineer an open-source software and a custom binary using various diagnostic tools and security assessment techniques. The open-source software we have is OpenConnect, it's a VPN service that connects to secure web servers, it includes phases for authentication using certificates and connection using a standard webvpn cookie which runs in Linux. We were using tools like obj dump, ldd, nm, strace and VM table dump to analyze the binary we build. We also utilized the Scitools, Kaitai struct tool, arch studio to better visualize the binary. We initially fixed a single byte error in the custom binary and visualized it using tools like binvis.io and veles tools. Lastly, we have attached key screenshots to our task.

Index Terms— Binary, graph views, Reverse engineering, VPN, SSL, Entropy, Obj dump

1 PROJECT OVERVIEW

THE project included performing various reverse engineering tasks on 2 separate binary files. The first binary file is the open-source application we selected, CISCO OpenConnect VPN, which must be compiled and run in Linux. We were able to accomplish this by doing the following:

- Dumping the headers
- Listing the shared library dependencies
- Listing various symbols in the binary
- Dumping VM tables
- Diagnosing with Strace tool
- Constructing a top-level system architectural diagram
- Plotting various views like cluster call butter-fly graph and control flow graph
- Analyzing the software for new vulnerabilities and the approach for discovering the same

The second binary file, that was provided, had to be initially corrected and it was reverse-engineered through visualizing tools like:

- Binvis
- Veles

We were able to visualize the reverse engineer both the binaries and the results from the various tools provided a wholesome picture of both the software in a very different light. This helped us in understanding how to discover the vulnerabilities using reverse engineering methodology.

1.1 CISCO OpenConnect Overview

The VPN is based on the standard HTTPS and DTLS protocols. The VPN connects to the secure web server and authenticates using certificates and/or arbitrary web forms, after which we will be rewarded with a standard HTTP cookie named webvpn.

After authentication, the webvpn cookie will be used in the HTTP CONNECT request, which is passed through the traffic. Following which IP addresses and routing information are passed back and forth in the headers of that CONNECT request.

Since TCP over TCP is very suboptimal, the VPN al-so attempts to use UDP datagrams and will only actually pass traffic over the HTTPS connection if that fails. The UDP connectivity is done using Datagram TLS, which is supported by OpenSSL.

1.2 CISCO OpenConnect Architecture

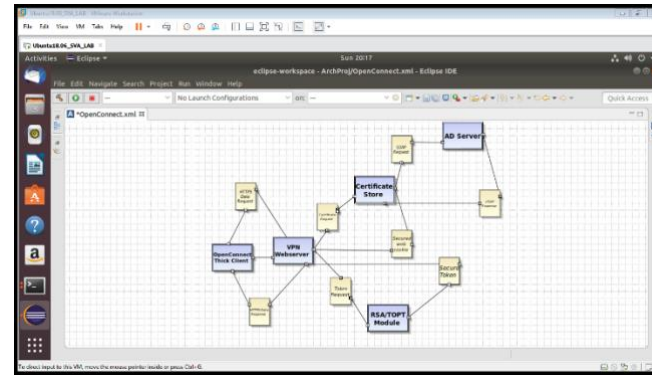


Figure 1 High Level Architecture of Openconnect

Open connect is an SSL VPN Client for secure communication. It connects through HTTP and SOCKS5 proxy including support for automatic configuration of proxy using libproxy library. It can detect both IPv6 and IPv4 addresses and routes.

The connection happens in two different phases: 1) HTTPS connection for users is authentication either by a certificate, password or secure id. Once authentication is complete, an authentication cookie is provided as a response to make the actual VPN Connection. 2) This cookie connects to a tunnel using HTTPS through which data packets can be passed over. A UDP tunnel can also be configured in the case of UDP traffic, which can be disabled if required.

2 PROJECT EXECUTION

2.1 Open Source Project

We started initially by going through the documentation regarding CISCO OpenConnect VPN that is available on the official website and also in GitLab [https://gitlab.com/openconnect/openconnect]. We git cloned the code to our Linux system. We installed the pre-required libraries provided by the documentation. This was followed by configuring, installing and compiling we obtained the binary for open connect. We have documented the key steps associated with the building of the binary through which we were able to observe and identify the source language, compiler, and the OS. We used the Kaitai tool for decompiling the binary and was able to parse through the file which made it comprehensible.

2.2 OpenConnect Tools Reverse Engineering

We were able to obtain the relevant headers using the command $ objdump -h open connect, which displays summary information from the section headers of the object files. Then the shared linked libraries of the binary are obtained through the com-mand $ ldd open connect. The symbols from the binary executable are obtained through the command $ nm open connect.

We were able to further explore the functionality of the service by running the binary. We dumped the virtual memory tables associated with the binary using the command $ cat /proc/<pid>/maps where pid corresponds to process id of open connect. We also were able to diagnose the open connect using strace, $ strace -p <pid>, which intercepts and records all the system calls and signals which are called and received by the open connect respectively. Hence, we were able to document the key functionality of the VPN and the various interaction with the other system process. By these

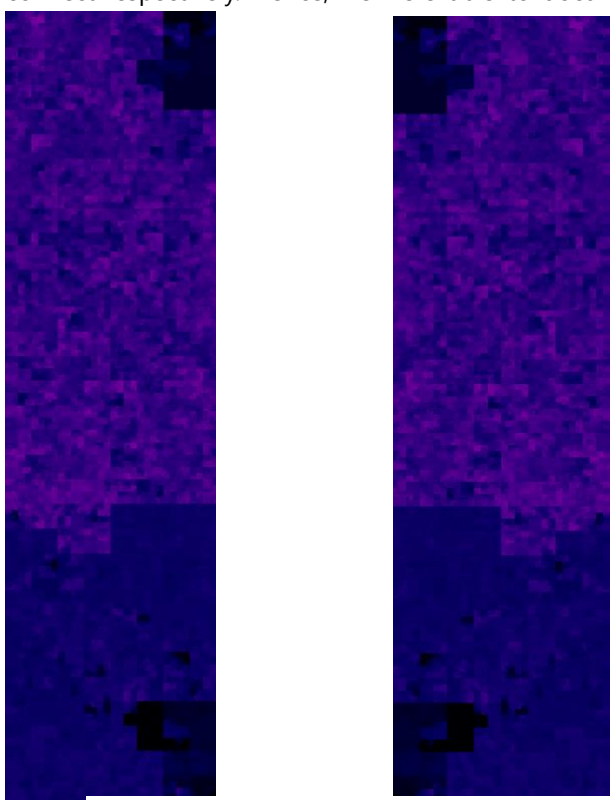
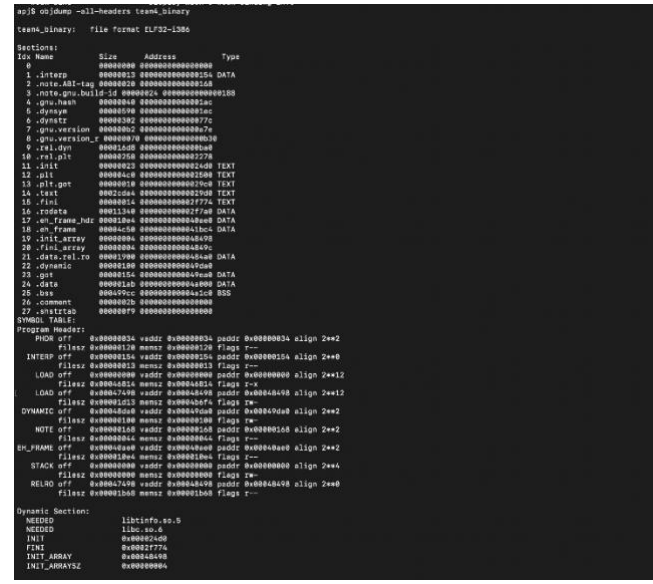


Figure 2 Entropy Visualization using binvis



```
asj$ objdump -all-headers team4_binary

team4_binary:   file format ELF32-i386

Sections:
Idx Name          Size      Address          Type
  0               00000000  0000000000000000
  1 .interp       00000013  0000000000000154 DATA
  2 .note.ABI-tag 00000020  0000000000000168
  3 .note.gnu.build-id 00000024 0000000000000188
  4 .gnu.hash     000000c8  00000000000001ac
  5 .dynsym       00000590  0000000000000274
  6 .dynstr       00000382  0000000000000804
  7 .gnu.version  000000b2  0000000000000b7e
  8 .gnu.version_r 00000070 0000000000000c30
  9 .rel.dyn      000016d8  0000000000000ca0
 10 .rel.plt      00000258  0000000000002378
 11 .init         00000023  00000000000025d0 TEXT
 12 .plt          000004c0  00000000000025f0 TEXT
 13 .plt.got      00000010  0000000000002ab0 TEXT
 14 .text         0002cda4  0000000000002ac0 TEXT
 15 .fini         00000014  000000000002f774 TEXT
 16 .rodata       00011340  000000000002f7a0 DATA
 17 .eh_frame_hdr 000010a4  0000000000040ae0 DATA
 18 .eh_frame     0000ac58  0000000000041bc4 DATA
 19 .init_array   00000004  000000000004849c
 20 .fini_array   00000004  00000000004849c
 21 .data.rel.ro  00001900  00000000004849e0 DATA
 22 .dynamic      00000100  0000000000049de0
 23 .got          00000154  0000000000049ee8 DATA
 24 .data         000001a0  0000000000049a000 DATA
 25 .bss          000499cc  0000000000049a1c0 BSS
 26 .comment      0000002b  0000000000000000
 27 .shstrtab     00000079  0000000000000000
SYMBOL TABLE:
Program Header:
    PHDR off    0x00000034 vaddr 0x00000034 paddr 0x00000034 align 2**2
         filesz 0x00000120 memsz 0x00000120 flags r--
  INTERP off    0x00000154 vaddr 0x00000154 paddr 0x00000154 align 2**0
         filesz 0x00000013 memsz 0x00000013 flags r--
    LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
         filesz 0x00041d14 memsz 0x00041d14 flags r-x
    LOAD off    0x00047498 vaddr 0x00048498 paddr 0x00048498 align 2**12
         filesz 0x00001d13 memsz 0x00004bf4 flags rw-
 DYNAMIC off    0x00048da0 vaddr 0x00049da0 paddr 0x00049da0 align 2**2
         filesz 0x00000100 memsz 0x00000100 flags rw-
    NOTE off    0x00000168 vaddr 0x00000168 paddr 0x00000168 align 2**2
         filesz 0x00000044 memsz 0x00000044 flags r--
EH_FRAME off    0x0004ae0 vaddr 0x00040ae0 paddr 0x00040ae0 align 2**2
         filesz 0x0001bc4 memsz 0x0001bc4 flags r--
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rw-
   RELRO off    0x00047498 vaddr 0x00048498 paddr 0x00048498 align 2**0
         filesz 0x00001b68 memsz 0x00001b68 flags r--

Dynamic Section:
  NEEDED               libtinfo.so.5
  NEEDED               libc.so.6
  INIT                 0x080025d0
  FINI                 0x0802f774
  INIT_ARRAY           0x0804849c
  INIT_ARRAYSZ         0x00000004
```

Figure 3 Obj dump Displaying headers

observations, we were able to also construct a top-level architecture diagram using Arch Studio, which clarified different parts of the software structure.

We were also able to plot various metrics associated with the binary by using the SciTool Understand. With the same tool, we were also able to get a visual representation of the binary file with plots like cluster call butter-fly graph and control flow graph views that represent pictorial depiction between components and the logic flow. We were able to identify functions that handled the sanctifying of user input and possible vulnerabilities present in the software.

2.3 Custom Binary Executable

We started reverse engineering of the binary file using various visualization tools like binvis.io and veles tool. We initially fixed the single-byte error in the binary file by going through the headers. We were able to compare the entropy and file structure of both the erroneous and fixed binary files using binvis. We also used the veles to visualize and observe the layered and trigram view of both the binaries and noted the difference. We also used the tools from previous tasks to identify the behavior, headers, and executables of both binary files.

2.4 Binvis, Veles Tools Execution Summary

As only a single byte is changed in our given binary, we won't be able to notice significant changes in both entropy and Hilbert curve visualization. Also, from Binvis entropy visualization, if there is a block with a dark shade of pink in the Hilbert visualization, this is a strong indication of the possible presence of packed/compressed or cryptographic material.

From the binary given, there is a lot of ASCII section and it comprises of several strings which we were able to extract using the strings command. In the given binary, there are no significant pink patches, from this we can conclude

that there isn't significant compressed content inside the binary.

Upon analyzing the given binary file, we understood that the manipulated byte is present in the header and to correct the manipulated byte, we iterated the first 16 bytes of the header with different characters and upon replacing the header byte A with E, the resultant binary is a valid format and it is re-confirmed with the results obtained from other reversing tools. Manipulated byte is corrected.

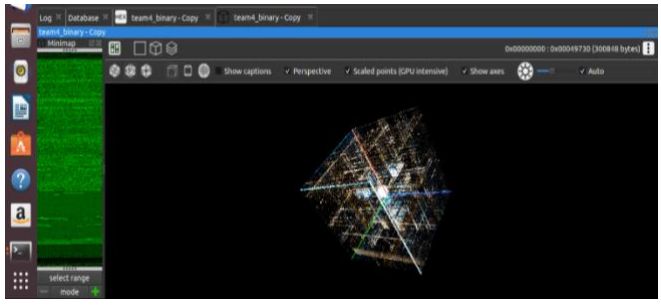


Figure 4 3D Visualization using veles tool to identify File type

When we opened the binary in VELES, by observing the tri-gram view of the file, we can see that the distinct perpedicular line corresponds to the basic pattern of the compiled binary. The cuboid in the center of the structure represents the ASCII characters which indicate that the file contained some RAW text. We also analyzed the trigram view of the binary file again and selected the range of machine code to find multiple distinctive perpendicular bars. The x86 32-bit machine code has characteristics of distinct perpendicular bars. But since default operand size for many instructions in 64-bit mode is 32 bits, hence we can conclude that this binary is an x86 64-bit machine code.

3 Accomplishments

We were able to understand the binary in more details after reading the man pages for several Linux commands like ldd, objdump and dumping the process VM Tables. Also, we were able to successfully compile the program using the steps outlined in the project website and in the process we encountered several issues, and after referring online we solved the issues after installing build tools in the Linux system and able to make the build file and install it on the Linux system. We were able to launch the process and able to extract the Process id and use cat/proc/<SOME_PID>/maps to obtain the VM Tables which are very useful resource to understand the dynamic linking inside the binary and also we used strace system command to see how the binary execution takes place internally and able to read the strace command output and identified several sys calls and interrupts. Several online resources and man pages gave us valuable information on how to use these tools to successfully analyze the binary. This is a great achievement for our team as we were able to successfully build a binary and reverse engineer it using several industry grade tools. While performing the reverse engineering, we were able to think like a attacker and execute several commands just like how an attacker tries to obtain information from a given binary including headers dump which gave us valuable information about various sections of the binary including heap, stack, code and text, this information can be used to launch attacks such as Buffer Overflow, shellshock and dirty cow vulnerabilities. Also, we analyzed the binary source code of the OpenConnect VPN Tool and identified a top-level system architecture and identified main system functions and various threat levels associated with them. While using the Understand tool, we learned about the code level metrics like no of executable lines, no of functions, etc and able to successfully identify the dependency graph and call flow graph of the OpenConnect VPN Tool.

4 Lessons Learned

We had a few troubles initially to get the binary of the software, mainly because of the pre-requisite libraries we had to install beforehand. But after some time, we were able to quickly install all required libraries and build our

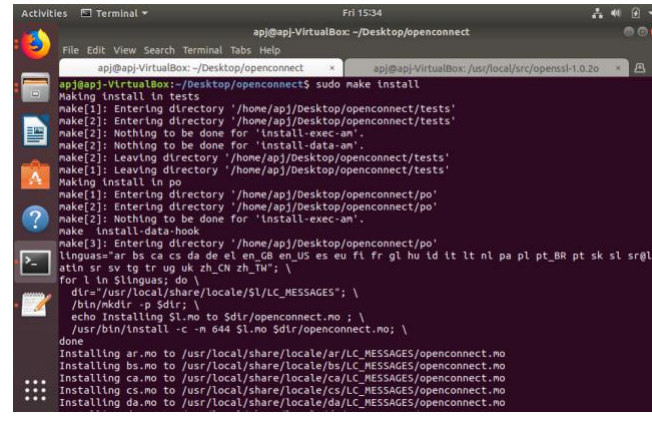


Figure 5 Successful Compilation of the binary.

binary as shown in Figure 5.

We learned from this project how demanding it is to build software like OpenConnect can be. We had to go through the source code to understand the in-depth functionalities, various input sanitation, several important system calls, etc.

Acknowledgment

The authors wish to thank Reuben Johnston for his support during this project.

References

[1] OpenConnect Tool Build Instructions, "https://www.infradead.org/openconnect/building.html", 2019

[2] Infradeed, Installing Open connect VPN vpnc script and its

configuration,https://www.infradead.org/openconnect/vpnc-script.html, 2019

[3] Binvis, “A browser-based tool for visualizing binary data,” https://corte.si/posts/binvis/announce/index.html. 2015.

[4] Understand - Scitools, “A tool to visualize your code and information about functions, classes and variables and its usage, https://scitools.com/features/

[5] Arch studio, "Getting started with Arch studio and create new architecture,"http://isr.uci.edu/projects/archstudio/getting-started.html

[6] Linux Audit, The 101 of the ELF files on Linux, https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/

[7] Reverse Engineering with Kaitai struct, https://gettocode.com/2017/09/18/reverse-engineering-with-kaitai-struct/, 2017

[8] Kenneth G. Hartman, “Calculate File Entropy of the Binary Files”, https://kennethghartman.com/calculate-file-entropy/, 2013

[9] Lysne O. (2018) Reverse Engineering of Code. In: The Huawei and Snowden Questions. Simula SpringerBriefs on Computing, vol 4. Springer, Cham

[10] Tyson, Jeff. "How Virtual private networks work." Howstuffworks,(Jul. 12, 2005) (2001).