Reverse Engineering Unknown Protocols

Tonimir Kišasondi; Željko Hutinski,Vesna Dušak
Faculty of organization and informatics
{tkisason, zhutinsk, vdusak}@foi.hr

Abstract. *In this work we will present our method for analyzing and reverse engineering unknown or proprietary protocols for the purpose of analyzing the protocol security or gaining more insight into the protocols inner operation. We created a method that can be executed manually or can be packed into a fully automatic algorithm.*

Keywords. Reverse engineering, protocol analysis, black box analysis

1. Introduction : The need for protocol reverse engineering

When installing or implementing a third party solution that employs proprietary protocols for communication, there is a need to analyze and assess the strength of the implementing communication protocol at the application layer regarding to the manufacturers claims. Since only some manufacturers which are a minority in todays corporate realm specify the protocol and give out the whole documentation that is needed for an independent audit or an implementation with a in-house solution that needs to interconnect with the original manufacturers solution. For example, one such situation is if the manufacturer is not available because he went out of business.

Therefore, with security auditing primarily in mind, we developed a method for analyzing closed-source protocols and simply trying to assert as much as possible information from only observing and analyzing the outputs with regard to manipulation of the input data of the analyzed object in a systematic way, where we analyze the object as a "black box".

The complexity of testing such systems is high because we don't know anything about the internal functions or internal interactions of the observed system, like the protocol sentence semantics and syntax of the raw data we receive on the output. But we can soften this situation because we know something about the part of the input we send in the system. Those parts can be user defined (usernames, passwords, data) or some internal state that we can assert or edit from the device like time, date, timezones, ip address, current state and similar settings that can be changed or simply read from the machine. If we are dealing with unencrypted output which is mostly the case from our practice, we can disassemble the output data with our described method. If the data is encrypted, we can also assert some information about the properties of such encryption. Also, there are a number of common design patterns that can be assumed and that aid us in our analysis, as we will show in the following chapters.

The property we need to note is that we cannot effectively and hastily analyze a protocol with a single output. This assumption leads to a analyzing process which is difficult and unfeasible. We need to manipulate the device or software to give as much data output possible to create a valid dataset so we can observe and note emerging patterns.

2. Analyzing content and patterns

When beginning the analysis, we must note as much data as we can from the starting state of the object (meaning a device, software output, unknown network protocol or anything which we analyze and want to reverse and understand it's internal functioning). We must note as much data as we can, and in case of changing data, like time for example we must be able to note that data also. An analysis starting state is the state when we are starting our analysis, and for the collection of the starting data, we can even manipulate the object or physically disassemble it. For example, such starting data can be and is not limited to:

1. Time and date
   a) Time and date formats
2. Timezones
   a) Timezone formats
3. Default users and user ID's
4. Serial numbers
   a) Device serial number
   b) All integrated circuit, CPU's and other serial number or observed ID data.
5. Internal settings or related settings
   a) IP or protocol addresses
   b) Open TCP/UDP ports
   c) ICMP responses
6. Software version
   a) Firmware versions and revisions
   b) Internal, proprietary versions
   c) Standard software or firmware versions

7. All variable custom input data or variables
8. Encryption hardware, accelerators or IC's that help in facilitating and indentifying encryption mechanisms.
9. Calculating the whole input data size (approximation) and Measuring output data size and correlating to known input elements.
10. Fragmentation in other protocols (IP, TCP)
    a) Specific flags in different protocol implementations
    b) Responses to malformed packets
11. Additional properties and anomalies in other protocols and other layers because the object uses the proprietary protocol stack
12. TCP stack ISN sequence graphing (Zalewskian attractors [1])
13. Possible design patterns or errors
    a) Usage of open-source implementations
    b) Known proprietary implementations
    c) Previous and similar vendor or implementation vulnerabilities
14. Implementation specifics and uniqueness
15. Port scanning the hardware device and trying to fingerprint all running services.
    a) Standard penetration testing methods if the device is running services on TCP/UDP

Also, it's important to note that this is not the only data we can assert from the starting state, and depending on the object, more or less data that is or is not stated here can be available, it is important to try and obtain and identify as much data as we can from the starting state, even if some piece of data seems irrelevant as that starting data will aid us later in our efforts to reverse the protocol.

3. Pattern and content identification

After the initial data collection from the starting state, which is a snapshot of the system at its steady state we can start collecting data for the initial analysis. We select an initial set of user passed data to the object and simply run 10 or 20 runs which we record as the exit protocol dump. It is important to note that it is helpful that the picked set has as many distinct and ordered elements as possible. Our suggestions recommend the usage of a single letter or number per each field, so that the same letters or numbers can be easily identified on the output. Protocol sentence output captures can be done with Wireshark [2], Netcat [3], tcpdump [4], logical analyzers connected to the data buses and other methods depending on the analyzed object. After we have collected about 10 or 20 dumps (grouped in timed intervals and spaced in distinct patterns (5 outputs in within 1 minute, 5 outputs after 10 minutes and 5 after one hour for example) we can analyze the internal patterns.

Our preferred method of comparison is with comparison of hex and/or binary representations of the output. Since those outputs all have the same internal and user supplied data except timestamps which we simplified by mapping the outputs within predictable and regular time intervals that can be easily ascertained. Hex or binary dumps can be analyzed in two ways depending on the endianness of bit/byte data. The problem of endianness depends on the creators general design of the object. Network protocols prefer the big-endian order, but some designers prefer creating storing their data in little-endian order. So, if the standard big-endian order analysis cannot deliver any results, we can try analyzing the whole data in little-endian order, which is usually the case when we know that the data is not encrypted and we cannot ascertain any information from the protocol.

Assuming the default big-endian order, we can group all our outputs vertically and simply simplify the whole protocol by simply observing the changes in the substrings. For example, we will show a subset of a hex dump from an simple password logging aplications analyzed protocol to illustrate the point:

Aplication layer protocol hex dump:
1af020070921012001330000004d25f
1af020070921012001340000004d25f
1af020070921012101350000004d25f
1af020070921012101360000004d25f
1af020070921012101360000004d25f

Table 1. App layer protocol hex dump

While with grouping we simplify it in:

A	B	C	D
1af02007092101	200	133	000004d25f
	200	134	
	210	135	
	210	136	
	210	137	

Table 2. Vertical simplification of outputs for Table 1.

We can see that from the output, that when we group and simplify the resulting hex dump, that we actually have 2 changeable parameters (marked with B and C in Table 2) in our output hex dumps. The next step in simplifying the output is to see how the output parameters behave. In this example, we see that the parameter B is positive increasing, although not linearly, and that the parameter C is linearly increasing. This gives us the variant that C is most likely some counter. To analyze B or any other such

unrelated and unknown parameter we can try two methods: the first one is trying to parse the output protocol sentence and try to find a corresponding match between the encoded or plaintext input which we collected in our starting state and the output sentence. Such searching can be done manually which is the most thorough and slowest solution or more hastily and automatically with classic string searching algorithms like Knuth-Morris-Pratt and similar ones in a effective manner which we will describe later. If we cannot find a plaintext match, we can try and encode the data with various encodings. All encodings specified in this work are common and well known, and are well documented in their RFC's and their corresponding standards. Some of those encodings are:

1. Timestamps
    1. Plain, unencoded time
        1. Standard “ddmmyyyy” variants
        2. Reverse “yyyymmdd” variants
    2. UNIX/POSIX timestamps
    3. UTC
    4. TAI / TAI64NA
    5. GPS encoded time (NMEA time or sentences)
    6. Other 32 or 64 bit second encodings
2. Representations, encodings and encoders
    1. ASCII, unicode and others character encodings
    2. Base64
    3. uuencode
    4. yEnc
    5. MIME

If the segment we are trying to understand is located at the end of the protocol sentence or is totally random with maximum entropy with the respect to the observed segment we can check redundancies or error corrections from the rest of the protocol segment. Such redundancy or error correction codes can be:

1. CRC / CRC32
2. Checksums
3. Hamming codes
4. Turbo codes
5. Space/Time codes
6. Reed Solomon codes

Also, if the analyzed entropy from the segment is maximal then we can also assume that some kind of encryption or hashing is used. Cryptographic mechanisms can be usually identified by perfect randomness, high entropy and some predictable block size like 128, 256, 512 bits and any multiplier of such blocks. Encryption can be possibly cracked if we can supply information to the analyzed object, therefore creating an partial adaptive chosen plaintext attack. Doing cryptoanalytical work on an fully unknown protocol, with partial known plaintext and known ciphertext is extremely difficult, so in case of a hardware object, disassembly and trying to read the keys from EEPROM, flash memory or directly from the data bus with a logic analyzer would likely be more feasible and is usually more efficient. For software objects, classic disassembly or software reversing would be feasible if obfuscation techniques haven't been used or if weak obfuscation is used. Chosen plaintext attacks can be mounted if the object uses authentication elements like passwords, PIN's and other similar content. That way, we can hash the authentication element and see if the output contains the hash result. Also, hashes can be used for data integrity. Since it is less likely that a protocol will use hash functions for data integrity, but we can try to mount a chosen plaintext attack by hashing the authentication element at the input and then searching the output sentence for the hash. This can usually be the case if the protocol transmits a large portion of data and then verifies the integrity of a point to point transfer with a hash function. Such popular hash algorithms are:

1. MD family
    1. MD2
    2. MD4
    3. MD5
2. SHA family
    1. SHA-1
    2. SHA-2
    3. SHA-256
    4. SHA-384
    5. SHA-512
3. Adler32
4. Haval
5. RipeMD family
    1. RipeMD-128
    2. RipeMD-160

Also, if full protocol layer encryption is used, it will be mostly implemented with some wide known open source method like SSL or TLS. Also an effort in reversing can focus on trying to crack the protocol if a weak implementation is used like SSL version 2, or if a random generator used is taking his entropy pool from a limited source which can be the case since embedded systems usually don't rely on good random generators but mostly on most simple or quick ones.

It is important to note, that all mentioned patterns, algorithms and encodings are not the only ones that can be present in an analyzed protocol dump. The mentioned patterns and algorithms here are the most common from our practice and experience. If comparison to standard patterns cannot yield any information we can try several things like treating the parameter if it is increasing per each sentence output as an mathematical sequence and try to identify patterns by calculating differences in

increases or decreases. Another helpful method can be graphical plotting the parameter in two or three dimensional space with some plotting program like gnuplot [5]. Also, attractor analysis like in [1] can be extremely helpful. Other helpful insight can be gained with calculation of entropy per substrings in the whole output. That way we can see which part of the output has the maximum randomness and can be assumed to be a redundancy code or other encoding.

Guided by the example shown on table 2. We can see that C is a counter and from A and B we can get the timestamp: 200709210120 which is 2007.09.21.01:21(yyyy.mm.dd.hh.mm format). Also with searching for the input data (password: 1234) we can see that hex format of the data is: 0x4d2 which we can find in the string with indexes of 26 to 28. That way we have a partial reverse, also we can try to search for other input data to try and do a full reverse.

4. Reverse protocol modeling

Since we know how to ascertain the output protocol format, for the purpose of interconnection with other systems we usually need to reverse the whole protocol to a useful level. Since semantics are clearly not enough, we need to reverse the entire protocol structure. Protocols behave differently to signal various states they can be found in. Those states are usually:

1. Boot up phase (B)
2. Connection start (setup phase) (St)
3. Data transfer (transfer phase) (Tr)
4. Signaling phase (special cases) (Sg)
5. Connection closing (teardown phase) (C)
6. Shutdown phase (E)

We can show the relation of those common protocol phases with a state simple diagram:

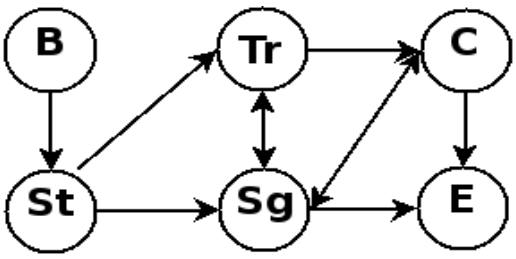

Diagram 1. Generic protocol states

States shown in diagram 1 usually define the states most common or proprietary protocols can find themselves in. Some protocols do not use all states, but most analyzed protocols can group their outputs based on the shown generic state diagram. While reversing, if we suddenly get a different output that is different from the rest of the output sentences, we can assume that we entered a different protocol state. We can analyze states in the same way as we analyzed outputs the only difference is that we have to acquire multiple outputs per same state like multiple boots, starts, teardowns and errors so we can ascertain patterns from grouped states. That way, with fully identified patterns with every protocol state we can recreate the protocol state machine and make an interconnecting service or the audit we are doing can be more complete. Of course, reversing the protocol states and the meaning of specific fields in the output doesn't need to be full or complete, it is only important that the reverse either finds enough flaws that state the protocol is not fit for use, or enough knowledge that we can create our interconnecting service.

5. Automating the process into an algorithm

From the concepts outlined in this paper, we can simply pack the methods into an automatic algorithm. Since manual analysis does not require any automatic input or output our suggested procedure can be done manually. Packing the methods into an fully automatic algorithm is mostly straight forward and can be used effectively in combinations with fuzzers or as a part of a fuzzer and other tools to help with protocol security testing. If we are analyzing software, input data and output data collection and insertion can be automated with ease. If we are analyzing hardware objects, the problem is with the input data is internally regulated like timestamps. Internal hardware tapping is difficult because it usually requires the modification to the object or complex tapping or directly soldering taps to data buses. If we wanted to automate a great deal of the process we would use the following algorithm:

First we need to tap into all inputs and outputs in the object, so that we are able to send and receive data from the object. Next, we input all static starting data shown in part 2 of this paper, including clock synchronization from the analyzed object to facilitate easier variable and timestamp identification. After that, we choose a distinct set for our input variables of input data, and request multiple outputs. Our recommendation is 10, 20 or more. After that, we simply do a vertical difference based on the outputs we have, which helps us in identifying static data. After that, we start testing all known encodings as shown in part 2 with our input data variables. It is helpful if our input data is unique and specific so that the identification is simple. Searching the output protocol sentence can done with standard string searching algorithms. The different outputs that can't be identified can be clustered into groups for analysis as different states. The algorithm for pattern matching can be applied multiple times on different states, in order to ascertain the semantics of the outputs. Also,

the algorithms effectiveness is determined by the base of known encodings and representations because if we have a bigger base then we can identify the encoding or representation which the creators of the protocol used. Therefore we need to try and populate the base with as much known algorithms as possible. Also if we didn't get any senseful results, it is helpful that we try to analyze the whole protocol in different endian order, which is usually little endian order. Application and optimization on multiple core processors is simple, because string searching can be easily parallelized.

6. Conclusion

In this paper we shown the general method which can help with protocol reverse engineering. Since most scripting languages like perl or ruby have most encodings, hashing functions available in libraries, and all functions are well documented we recommend implementing the protocol dissector in a scripting language. A simple dissector for protocol dumps stored in files and for extraction of passwords from example in part 3 can be done in perl using the unpack and hex functions with ease. Also, protocol dissectors can be written for popular applications like Wireshark [2].

Also, with the rising popularity of internet services and internet enabled devices, we believe that independent security auditing of closed source protocols and devices will become more and more popular, because the users will want to know what kind of device they are implementing in their network and how will that device impact their whole network security, which in a general picture will force the developers to take interest into placing security as one of the foremost concerns in their designs.

7. References

[1] http://lcamtuf.coredump.cx/newtcp/ (Accesed 10.6.2008)
[2] http://www.wireshark.org/ (Accesed 10.6.2008)
[3] http://netcat.sourceforge.net/ (Accesed 10.6.2008)
[4] www.tcpdump.org (Accesed 10.6.2008)
[5] www.gnuplot.info (Accesed 10.6.2008)
[6] RFC 1319
[7] RFC 1320
[8] RFC 1321
[9] FIPS 180-2
[10] RFC 1950
[11] http://labs.calyptix.com/haval.php (Accesed 10.6.2008)