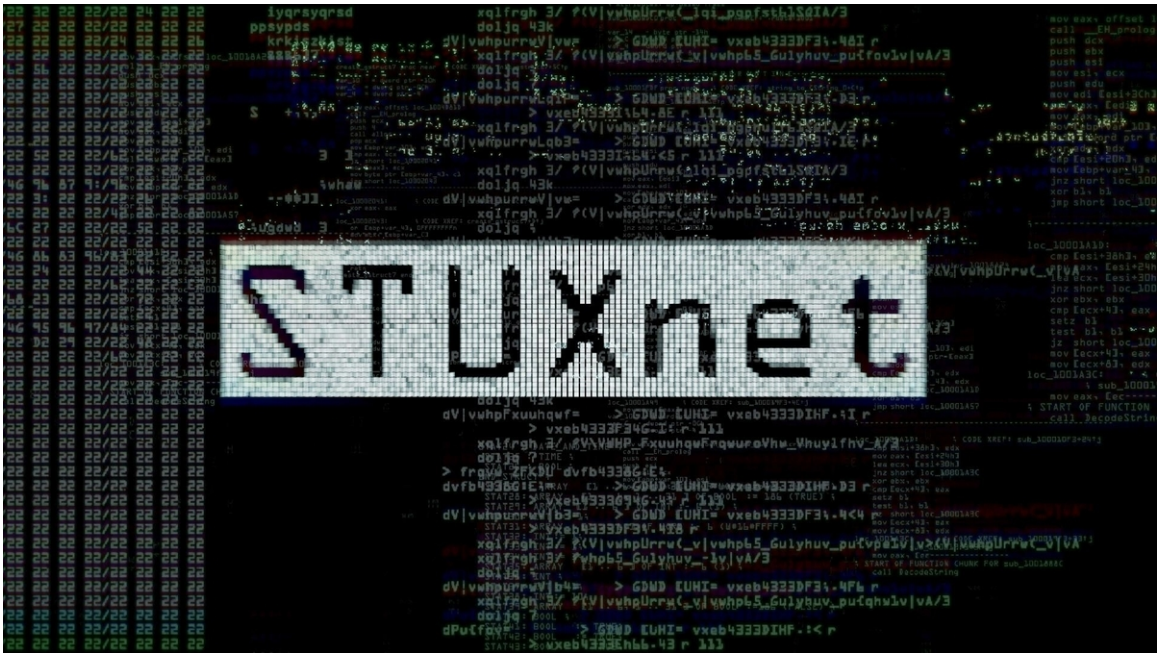
Northern Arizona University

Cybersecurity

CYB 410 - Software Security

Analyzing and Reporting on Stuxnet

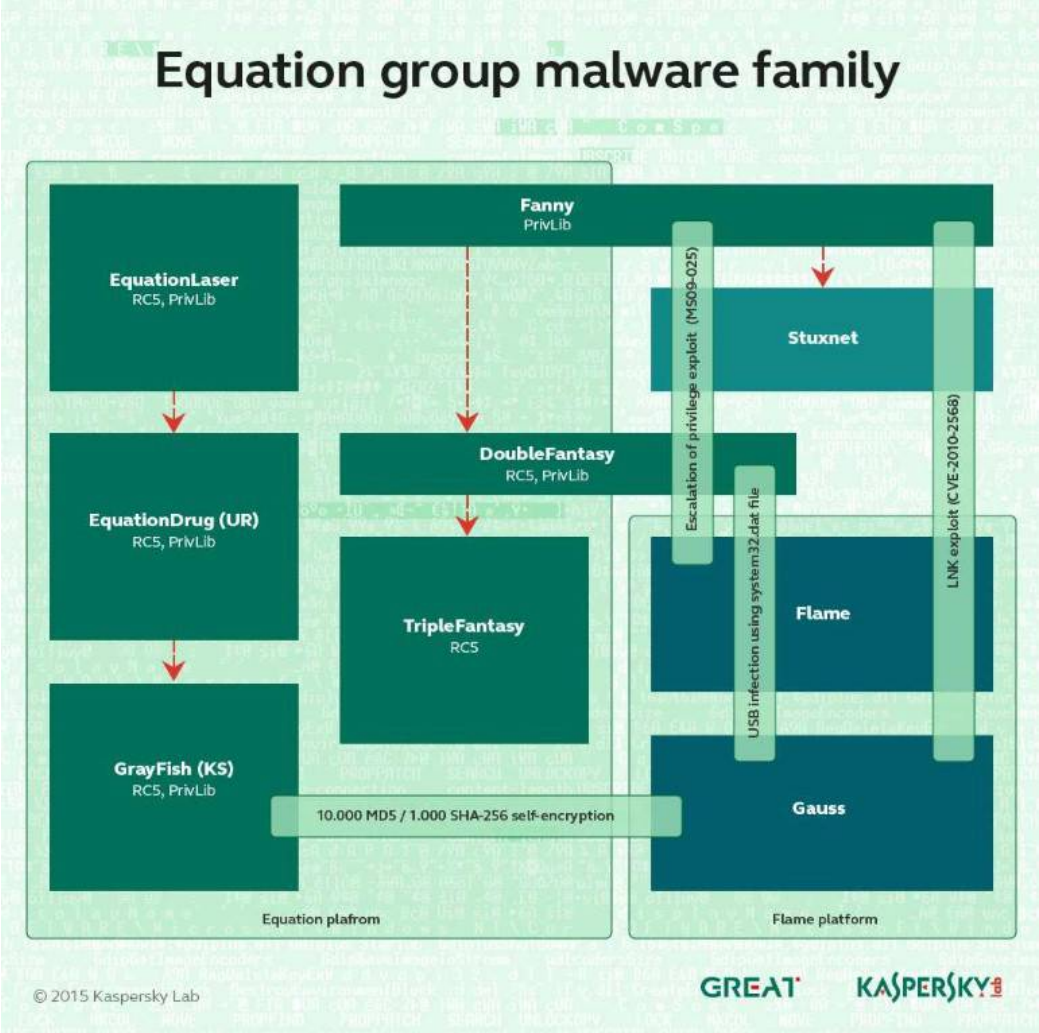


Author:
Akiel Aries

Professor:
Prof. Sareh Assiri

December 7, 2022

1 Overview



Stuxnet is a computer worm first identified in 2010 that was originally implemented to target Iranian governments nuclear facilities. Just like most virus', they mutate and spawn new strains often more dangerous than their predecessor. The original infection succeeded in targeting PLCs (Programmable Logic Controller) which are used in a plethora of manufacturing related processes. For example, PLCs are used in robotic machinery seen in assembly lines at manufacturing plants. The point of these controllers is to be secure, reliable, and useful fault diagnostics.
It is no wonder why Stuxnet was a large threat in the InfoSec industry. It is of popular belief that this is the first computer worm capable of affecting hardware and its peripherals as most computer worms are built as software exploits. The origin of the worm is cloudy however, according to reports from cybersecurity research firms such as Kapersky Lab, Symantec, and many more, the origin supposedly started as a nation-based collaboration with the United States and Isreal to target the Iranian Nuculear Program. The worm went on to destroy 1/5 of Iran's nuclear centrifuges, an important component in the process of enriching collected uranium, infecting

above 200,000 thousand devices and went on to severely degrade the performance of over 1,000 machines themselves. What lead researchers to believe the bug was nation state sponsored was the lack of infections of other users of the Seimens based software. This was false according toe Eugene Kapersky himself as he said a Russian neuclear power plant was also infected but not affected since it lacked any access to the public-facing internet. The common belief is that the group behind the attack is a nation state sponsored team by the name of Equation Group with reported ties to the USA's National Security Agency (NSA) which is also believed to be a part of over 500 computer infections. When it comes to violation of the CIA triad, this becomes somewhat murky territory as the attack was supposedly carried out the very own United States Government and the infected party was exploited by means of an air gap, meaning the bot's initial attack vector is physical. At a glance, all aspects of the CIA triad were violated as the confidentiality of the attack was not contained per leaked documents along with whistleblowing (allegedly), integrity was not upheld by the nation or said party involved, and availability was not present.
The repository that this report will be base the code off of is located here: https://github.com/research-virus/stuxnet

2 Fuzzing

The stuxnet virus makes use of infecting RPC (Remote procedure call) servers based on the different type of call types that are made. The servers are split into two components one managing local RPC calls and the other for managing remote calls. The calls that were sniffed were services.exe for local calls.
The bot makes use of exploited Windows systems on a very low level allowing the attacker to maniuplate portions of the Operating System perhaps not meant to be dealt with by regular users. Stuxnet makes effective use of exploiting at the time Windows inseucre digital certificate services. Stuxnet makes use of exploiting asynchronous encryption methods utilized by said certificate services, this allows Stuxnet to infect and spread by spawning processes different threads. To look into this I had found an example of simple asynchronous encryption in C++, allowing for decription from the CLI as well. See here: https://github.com/galets/oneway-cpp
When Fuzzing the program I had attempted for many hours to install deepstate produced by trailofbits, but failed in every way including within a Docker container. To make use of fuzzing the `async_encrypt.cpp` file I made use of AFL and AFL++. Using AFL++ and getting it installed was quite the process but since my reinterpretation of an asynchronous encryption algorithm accepted argc and argv paramaters I had to do some research on how this could be done. I went ahead and found a header file I could easily use along with some macros to initialize argv fuzzing with the default argv[0] set to the compiled binary itself.
Fuzzing the code itself took some work. I had created a Makefile for easier compiliation with the following contents:

```
oneway-AFL-COMP:
AFL_SKIP_CPUFREQ=1 AFL_HARDEN=1 afl-g++ -o \
```

```
oneway_AFL oneway.cpp -lssl -lcrypto

oneway-AFL-RUN:
AFL_SKIP_CPUFREQ=1 afl-fuzz -i in -o out ./oneway_AFL
```

the `AFL_SKIP_CPUFREQ` initializer skips the error that may pop up of innefecient CPU cycles. I chose to ignore this since the program I am testing is not large and I do indeed have suffecient CPU cycles. Compiling the code with afl-g++ allows for us to compile the code with instrumentation to be recognized by the fuzzer itself. To fuzz the code itself I had created an `in/` directory with test inputs to fuzz against (mainly just test cases of generating hashes in a asynchronous way) and generates an `out` directory with result of our fuzzer. This can be ran indefinitely and we can monitor the result as it goes. Here is an example of AFL++ running for about 25 minutes with the following results:

A number of 1 unique timeout was produced really being the only significant result. Perhaps if I refactored the code (if it wasn't over 600 lines) with more time I could've produced more interesting result but programs that make use of properly implemented CLI arguments and flags are hard to fuzz against.