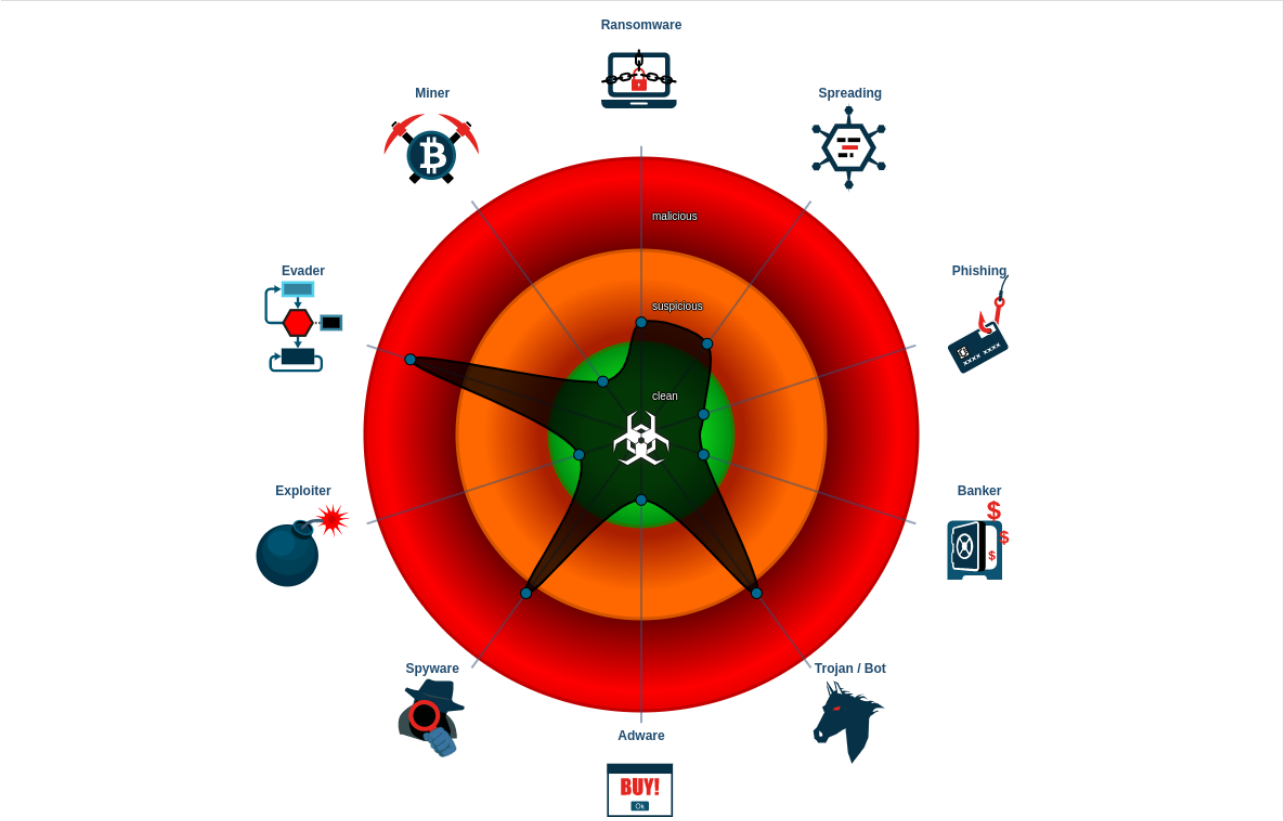
README.md

When messing with malware on your machine I figured I would either want to run it in a VM or make sure none of the prerequisites are enabled on the machine itself. This is where malware written in C or C++ thrive as languages like Python and Java need the runtime and interpreter already installed to meet that prerequisite. For this assignment I decided to take a look at the Zeus trojan malware pkg that initially gaining traction in the late 2000s when used to streal info from the US DoT. Trojans are known to be malware that behaves as legitimate programs. The bot would go on to infecting millions of computers and spawning any variants. The bot is aimed to target windows machines to extract banking informations using keystroke logging. This form of attack is a common example of a man-in-the-middle attack. The malware pkg performs infection via phishing schemes or drive-by downloads and infected some of the largest companies such as Bank of America, Oracle, Cisco, Amazon, etc. The trojan violates just about every principle in the CIA triad. Since confidential banking information was taken as well as money, confidentially and integrity are both violated while availability remains in questions. Patches to the malware offered from cybersecurity consultants is known to eradicate it from the infected machine, however strains of the virus were spawned remaining a constant battle.



I was able to find several versions of the "leaked" banking trojan but check out the version I had found from here!

https://github.com/touyachrist/evo-zeus

Although not the most dentrimental part of the source code, in a repo containing the Zeus source code this block minimally shows the entry point to the windows core API was created and giving root permission

```
void WINAPI entryPoint(void) {
  Mem::init();
  Console::init();
  Crypt::init();
  Core::init();

  CUI_DEFAULT_COMMANDLINE_HELPER;

  Core::uninit();
  Crypt::uninit();
  Console::uninit();
  Mem::uninit();

  CWA(kernel32, ExitProcess)(coreData.exitCode);
}
```

A similar version of this code I ran on my machine running kali linux, was able to give me root permissions as a regular user. See here (maybe it works for you):

```
int getuid(){
    return 0;
}
int geteuid(){
    return 0;
}
int getgid(){
    return 0;
}
int getegid(){
    return 0;
}
```

The bot has went thru many stages and versions with earlier versions of the bug had executable files hardcoded to one of these files

```
ntos.exe
oembios.exe
twext.exe
```

date being stored in the folowing dirs

```
System>\wsnpoem
System>\sysproc64
System>\twain_32
```

the next iteration and stored files in a single directory later found by security researchers, with another version storing executables in randomly named folders in app data.

The bot executes in the following stages

- Builder This malware comes in a kit usable by regular users with no technical knowledge, meaning the "owner" must deploy their own executables to who they wish to infect. This is done easily by using the builder in the tool kit. Each build will be unique to the infectee, due to some cryptographic implementations, there are uniquie keys generated for the configuration file embedded into the built .exe.
- Configuration Seperate from the build stage, contains the address to where the sniffed data is sent to in a series of blocks enables customization and hardcoding settings into the final binary. The config includes these sections:

```
StaticConfig
DynamicConfig
KeyLogger
WebFilters
WebDataFilters
WebFakes
```

Static and Dynamic Config both are dealing with the hardcoding of settings that eventually get executed at runtime. Here the questions of what is the target and destination get answered. In addition to the hardcoding of generic settings, dynamic features that imply additional comlexity such as redirection URLs of targets and destination addresses, URL masks, log disabling, sets of URLs performing Transaction Authenticaion Number (TAN) harvesting, as well as URL maks that contain corresponding HTML blocks injecting into webpages who match the Webinjects requests. The bot is responsible for running queries

Example of the config.txt file used to initate seen here: https://github.com/touyachrist/evo-zeus/blob/master/output/builder/config.txt

Example of webinjects.txt file used to targeting: https://github.com/touyachrist/evo-zeus/blob/master/source/other/webinjects.txt

- Execution The final executable file produced from build and configure (sounds like installing a C application) is finally deployed by the "owner" of the bug. If the .exe is produced with the same cofiguration and build settings the end results will usually vary in where the config file is stored.
- Server Finally the bot is deployed on a php-based server utilized by an abundance of php scripts that allows the deployer to monitor their results! This also serves as a sort of remote access type application where CMDs can be issued using this stage.
- Why wasn't it detected? From its initial release, the bot itself was made more complex and harder to detect for a number a reasons. What made things tricky was random naming of files to specific directories and in small sizes. Using checksums is monitoring bits transmitted at a higher rate than normal, letting professionals know of some potential issues. In earlier stages the bug transmitted files carelessly and copied them into the system dir. Later versions made use of ensuring dropped files were not to have the same checksum as the orignal.

The general function of the bug is to continuously spawn threads based on previous ones that go around crawling the infected devices hard drive. Doing so by embedding itself into system directories.

Here we can see in assembly code how finding new executables to download content. The config file is written into our registers and that same thread it is executed in will attempt to scrape for new .exe files for config to point to.

Here is an example of how the keylogging and screens scraping takes place by importing a hook to the API

```
user32!TranslateMessage
```

If the a left click is detected, a global flag is set within another PAI hook and another check is conducted to check if the user is visiting a location specified in the config file. Screen captures are then only taken when visiting what is specified by the creator. This uses win32 functions that can also be seen here: https://github.com/touyachrist/evo-zeus/blob/master/source/client/screenshot.cpp

```
HDC hDC = CreateCompatibleDC(0);
HBITMAP hBmp = CreateCompatibleBitmap(GetDC(0), screen_width,
screen_height);
SelectObject(hDC, hBmp);
BitBlt(hDC, 0, 0, screen_width, screen_height, x_coordinate,
y_coordinate, SRCCOPY);
```

Lets also take a look at the differences in a small implementation of the RC4 stream cipher in a early and more recent version

The difference being the second implementation is encrypting the config file a 0x100 byte key at build time. An extra layer scen in v2 adds more complexity by implementing a XOR decryption (last code block)

```
/*
 * Example of config file encryption seen in v1
 */

dataSize = size of data
dataIn = encrypted data
char b;
    for (i = 0; i < dataSize; i++) {
        dataOut[i] = 0;
    }
    for (i = 0; i < dataSize; i++) {
        b = dataIn[i];
        if ((i % 2) == 0) {
            b += 2 * i + 10;
        }
        else {
            b += 0xF9 - 2 * i;
        }
        dataOut[i] += b;
    }



/*
 * A look at v1.x encryption on config files
 */

int rc4_decrypt(unsigned char *in, unsigned long size,
    unsigned char *S, unsigned char *out) {
    int i, j, dataCount;
    i = j = dataCount = 0;
    unsigned char temp, rc4_byte;
    for (dataCount = 0; dataCount < size; dataCount++) {
```

```
            i = (i + 1) & 255;
            j = (j + S[i]) & 255;
            temp = S[j];
            S[j] = S[i];
            S[i] = temp;
            rc4_byte = S[(temp + S[j]) & 255];
            out[dataCount] = in[dataCount] ^ rc4_byte;
        }
        return dataCount;
    }
```

```
    for (m = (decSize-1); m >0; m--) {
        decData[m] = decData[m]^ decData[m-1];
        }
```

Further Look at RC4 Encryption + Static Analysis w/ CLang

The RC4 Stream Cipher was an encryption algorithm used in many Windows systems for a variety of applications. The Zeus bot exploited this algorithm and went under a series of patches (for example using logical operator XOR to swap instead of int/ char method)

We will take a look at how the cipher is implemented for small scale use (passing in a key, string, expecting a returned hash). The analysis on this isn't very exciting and was not implemented by the attackers but exploited by them. The errors in the RC4 algorithms are beyond what I beleive will be caught by a static analysis tool like CLang. Running the windows-based bug on my linux machine was troublesome with CLange so I had decided to look at a more broad issues that contributed to the bug.

So we will be looking at a minimal reproducable problem for catching bank statements from the system.

rc4-v0.c

ITER 1 :

```
$ ./rc4-v0 1 hello
087FAE01F8

KEY = 1
```

```
STRING = hello
HASH = 087FAE01F8
```

ITER 2 :

```
$ ./rc4-v0 1 HELLO
285F8E21D8

KEY = 1
STRING = HELLO
HASH = 285F8E21D8
```

ITER 3 :

```
$ ./rc4-v0 224 secure_software
F96FCDD6802CC1F80298D5C5439B26

KEY = 224
STRING = secure_software
HASH = F96FCDD6802CC1F80298D5C5439B26
```

ITER 4 :

```
$ ./rc4-v0 int too
28BEF0

KEY = int
STRING = too
HASH = 28BEF0
```

Produces different hash based on case-type

Static Analysis

```
clang --analyze rc4-v0.c
rc4-v0.c:72:33: warning: Result of 'malloc' is converted to a pointer
of type 'unsigned char', which is incompatible with sizeof operand type
int' [unix.MallocSizeof]
    unsigned char *ciphertext = malloc(sizeof(int) * strlen(argv[2]));
    ~~~~~~~~~~~~~~            ^~~~~~ ~~~~~~~~~~~

rc4-v0.c:79:12: warning: Potential leak of memory pointed to by
'ciphertext' [unix.Malloc]
```

```
    return 0;
           ^
```

When running clang --analyze on the file it warns us of our usage of malloc specifically with using char pointer types with malloc which takes type int as a paramater. We also get warned of our return value and a potential memory leak returning 0.

rc4_XOR-SWAP.c

This version uses the logic operator XOR to swap our elements instead of previous implementation with ints and chars

ITER 1 :

```
$ ./XOR-SWAP Key Plaintext
\xbb \xf3 \x16 \xe8 \xd9 \x40 \xaf \x0a \xd3

KEY = Key
STRING = Plaintext
HASH = \xbb \xf3 \x16 \xe8 \xd9 \x40 \xaf \x0a \xd3
```

ITER 2:

```
$ ./XOR-SWAP 1088 password
\x56 \x89 \x0d \x9f \x31 \xc0 \x49 \x1e %

KEY = 1088
STRING = password
HASH = \x56 \x89 \x0d \x9f \x31 \xc0 \x49 \x1e
```

Static Analysis

```
rc4_XOR-SWAP.c:73:24: warning: Result of 'malloc' is converted to a pointer
of type 'unsigned char', which is incompatible with sizeof operand type
'int' [unix.MallocSizeof]
    (unsigned char *)malloc(sizeof(int) * strlen(argv[2]));
                     ^~~~~~ ~~~~~~~~~~~~
```

We get the same issue as above when using malloc with pointer types of char and malloc taking in int.

rc4_XOR-SWAP-ASCII.c

This program runs the RC4 algorithm and converts it back to its original plaintext. Encoder and Decoder

```
$ ./XOR-SWAP-ASCII Key Plaintext
\xbb\xf3\x16\xe8\xd9\x40\xaf\x0a\xd3
encoded:�3V(@�J
\x50\x6c\x61\x69\x6e\x74\x65\x78\x74
decoded:Plaintext

KEY = Key
STRING = Plaintext
HASH = \xbb\xf3\x16\xe8\xd9\x40\xaf\x0a\xd3
```

Static Analysis

```
rc4_XOR-SWAP-ASCII.c:72:24: warning: Result of 'malloc' is converted to a poi
of type 'char', which is incompatible with sizeof operand type 'int' [unix.Ma
    char *tempstring = malloc(sizeof(int) * len);
    ~~~~~~             ^~~~~ ~~~~~~~~~~~

rc4_XOR-SWAP-ASCII.c:74:9: warning: 2nd function call argument is an uninitia
value [core.CallAndMessage]
        hext_to_ascii(tempstring,hex_encoded[i]);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

rc4_XOR-SWAP-ASCII.c:96:21: warning: Result of 'malloc' is converted to a poi
of type 'char', which is incompatible with sizeof operand type 'int' [unix.Ma
    char *dump_to = malloc(sizeof(int) * strlen(argv[2]));
    ~~~~~~          ^~~~~ ~~~~~~~~~~~

rc4_XOR-SWAP-ASCII.c:98:33: warning: Result of 'malloc' is converted to a poi
of type 'unsigned char', which is incompatible with sizeof operand type 'int'
    unsigned char *ciphertext = malloc(sizeof(int) * strlen(argv[2]));
    ~~~~~~~~~~~~~~              ^~~~~ ~~~~~~~~~~~
```

This program was a bit more interesting but of course is not the standard RC4 algorithm and features some functionality to convert or hash back to its original plaintext value passed in. Here we can see some of the same issues as our previous programs implementing the stream cipher plus one more issue with a call argument being an uninitialized value.

We can verify out hash results using the test vectors see here: https://en.wikipedia.org/wiki/RC4#Test_vectors

```
$ ./rc4-v0 Key Plaintext
BBF316E8D940AF0AD3

KEY = Key
STRING = Plaintext
HASH = BBF316E8D940AF0AD3
```

Final Notes

The biggest reason for Zeus to come about and still infect systems today is due to finding exploits within systems. Potential reasons for allowing something like this happen can vary, when developing code especially for such large and complex systems, it can be hard to think of all possible test cases. Such as bugs, vulnerabilities, etc. The bug exploits the use of the now insecure stream sipher RC4. Patching crypotgraphic vulnerabilities is no easy chore and requires some detialed knowledge of some complex topics like number theory and perhaps abstract and more theoretical facets of math. Implementing fixes for popular bugs is difficult especially in the case of something that gets replicated and reproduced in a new variation of the previous version. It is a constant cycle as a security engineer to stay ahead of exploiters while they stay ahead of you.