ZeusVM: Bits and Pieces

By Dennis Schwarz, Arbor Networks ASERT, August 2015

ZeusVM is a relatively new addition to the Zeus family [3] of malware. Like the other Zeus variants, it is a banking trojan ("banker") that focuses on stealing user credentials from financial institutions. Although recent attention has been on non-Zeus based bankers such as Neverquest [11] and Dyreza, ZeusVM is still a formidable threat. At the time of this writing, it is actively being developed and has implemented some interesting features such as a custom virtual machine and basic steganography. In addition, due to a recent leak of a builder program [7], the ability to create new ZeusVM campaigns is now in the hands of many more miscreants.

To foster a better understanding of ZeusVM, this paper examines some of the internals of the malware from a reverse engineer's perspective. While it doesn't cover every component, the visibility provided can help organizations better detect and protect from this threat.

Naming

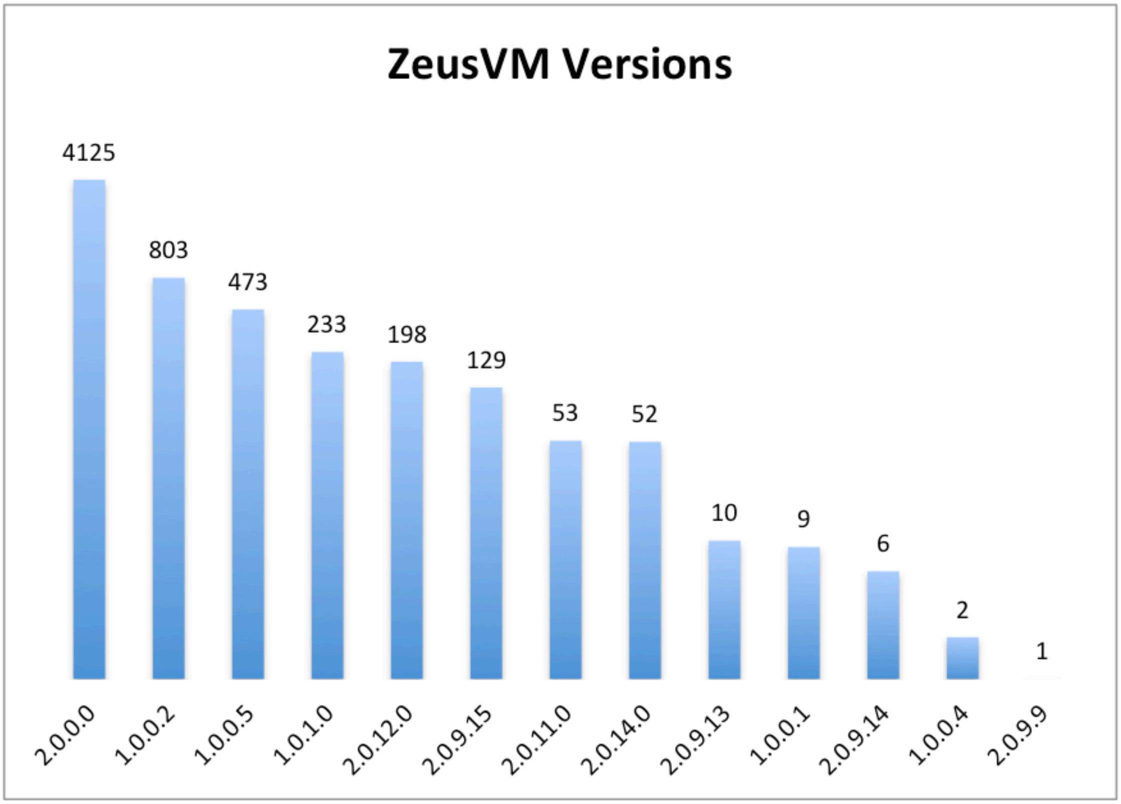
One of the first problems to tackle when reversing this malware family is figuring out whether to call it KINS or ZeusVM. The original batch of research [1] [2] from July 2013 was using the name KINS which is an acronym for Kasper Internet Non-Security. RSA discovered the name from an advertisement on an underground forum and Fox-IT got the name from a logo.

This early research chalked KINS up to be yet another in the long line of Zeus variants based on the leaked source code [4] of Zeus 2.0.8.9. Fox-IT was one of the first to note its most significant feature: a virtual machine used to decrypt a configuration file. Around October 2013, the source code [5] to KINS itself was leaked. While the leak confirmed KINS' lineage and virtual machine functionality, it likely spurred forks of the code base.

As more and more versions started appearing in the wild, some security researchers starting grouping any Zeus-based malware sample that uses the KINS virtual machine technology (including the original KINS) under the more descriptive family name of ZeusVM.

Versions

Grouping samples within the ZeusVM family can be done using their version numbers. The following figure shows the version distribution of ZeusVM samples within ASERT's malware zoo:



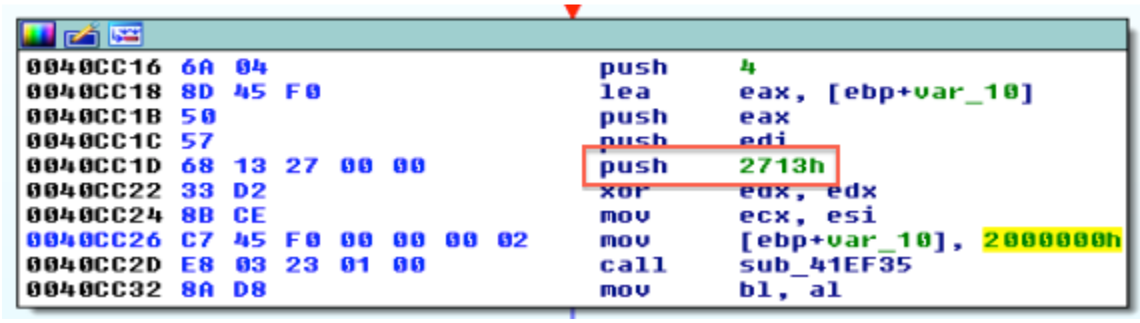
The format of the version is a.b.c.d where each letter is a number. This is consistent with other Zeus variants. Historically, per Zeus 2.0.8.9's leaked user manual [4], the version breaks down like this:

- a – "a complete change in the bot device"
- b – "major changes, that cause complete or partial incompatibility with previous versions"
- c – "bug fixes, improvements, adding features"
- d – "cleaning issue from antivirus for the current version a.b.c"

ZeusVM hasn't really followed this convention. Version-wise it hasn't really progressed linearly either. This is most likely due to multiple separate development threads. Based on ASERT's casual observations here is a rough progression of when versions were "active" in the wild:

- 2.0.9.13, 2.0.9.14, 2.0.9.15 – original samples from [2]
- 1.0.2.0 – from source code leak [5]
- 3.3.6.0, 4.6.9.0 – Zeus Maple [6]
- 1.0.0.1, 1.0.0.2, 1.0.0.4, 1.0.0.5 – RC4
- 2.0.0.0 – RC6 and the most popular version
- 2.0.11.0, 2.0.12.0, 2.0.14.0 – the most recent versions

Within the ZeusVM executable itself, the code block that contains the version can be found by searching for a PUSH or MOV instruction containing an immediate value of 0x2713 (highlighted in red):

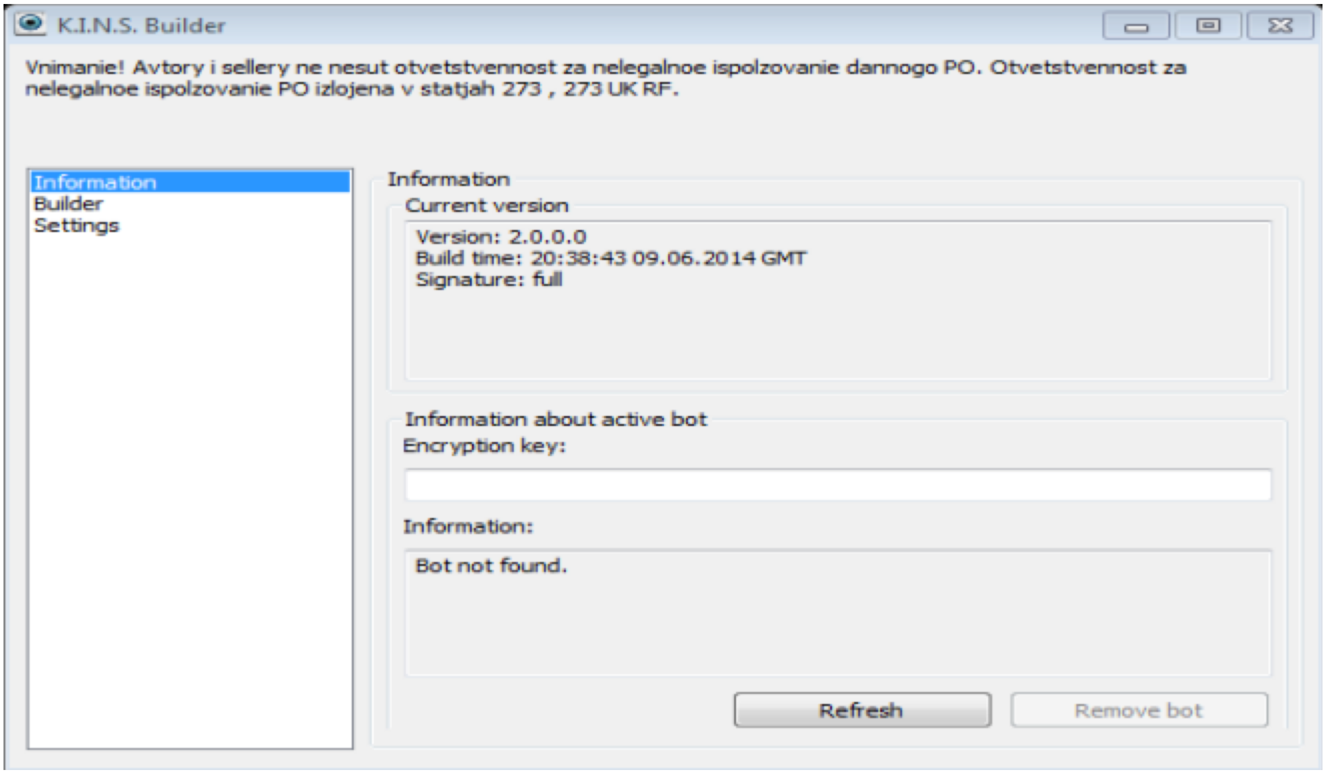


```
0040CC16 6A 04                  push    4
0040CC18 8D 45 F0               lea     eax, [ebp+var_10]
0040CC1B 50                     push    eax
0040CC1C 57                     push    edi
0040CC1D 68 13 27 00 00         push    2713h
0040CC22 33 D2                  xor     edx, edx
0040CC24 8B CE                  mov     ecx, esi
0040CC26 C7 45 F0 00 00 00 02   mov     [ebp+var_10], 2000000h
0040CC2D E8 03 23 01 00         call    sub_41EF35
0040CC32 8A D8                  mov     bl, al
```

The version is stored as a DWORD constant (highlighted in yellow). To convert this to the dotted decimal format: convert the value to hexadecimal, drop any leading zeros, starting from the left replace every second digit (zeros) with a dot, and then convert the remaining hex digits to decimal:

2000000 → 2.0.0.0

To narrow the scope, the rest of this reverse engineering analysis will focus on two specific ZeusVM versions: 2.0.0.0 and 2.0.14.0. The first was chosen due to its popularity. At the time of this research, ASERT's malware zoo contains 4125 unique samples of this version. The popularity and volume of this version will only continue to increase due to a builder being recently leaked:



The second, 2.0.14.0, was chosen because at the time of this research it was the most recent version that ASERT has seen in the wild.

Samples

The ZeusVM 2.0.0.0 sample used in this analysis has the following hashes:

MD5: b62c0477119c23af7ce308b913ed8514
SHA256: fd5cffe0c625a7603f13aacc118b2c60dd170e71fa214a45880d60c30b0c025c

This will be the default sample used through out this paper. When discussing version 2.0.14.0, the sample analyzed was:

MD5: d71c738c81962f392a60828aaeb2f6dd
SHA256: c5143a300fd4ee5d30000c41cf6e29dee106cabacc0708e92f37452867af6b60

Occasionally ZeusVM is contrasted with Citadel 1.3.5.1. The Citadel sample used has the following hashes:

MD5: 3763308503908aa4facf6e2897c2456b
SHA256: aebfcfa550f77f98e1ce625e9820ec6b09dba355e778053ad11adb8f481d975f

Base Config

After extracting the version from ZeusVM, the next issue to face is the base config. The base config is a hardcoded, sample-specific configuration data structure that is stored encrypted in the binary. Among other things, it stores the bot name, command and control URLs, and crypto keys.

While the layout and content of the data structure differs, the base config concept is shared among all Zeus variants. In addition, prior to ZeusVM, these variants all shared the same mechanism of decrypting the config.

Decrypting the Base Config in Other Zeus Variants

It is worthwhile to take a quick aside to review how other Zeus variants decrypt their base configs. This example will use a Citadel 1.3.5.1 sample, but is applicable to other variants such as Zeus Gameover, Ice IX, and the original Zeus.

The encrypted base config is stored hardcoded. Its size varies and for this particular sample is 1452 bytes:

Decrypting the data requires a key. The address of where the key is stored is generated at run-time and the length of the key is the same size as the encrypted base config (1452 bytes in this example):

The encryption algorithm is a basic XOR and can be described in Python like this:

```
plain = []

for offset, encrypted_byte in enumerate(encrypted_config):
    key_byte = xor_key[offset]
    plain_byte = ord(encrypted_byte) ^ ord(key_byte)
    plain.append(chr(plain_byte))
```

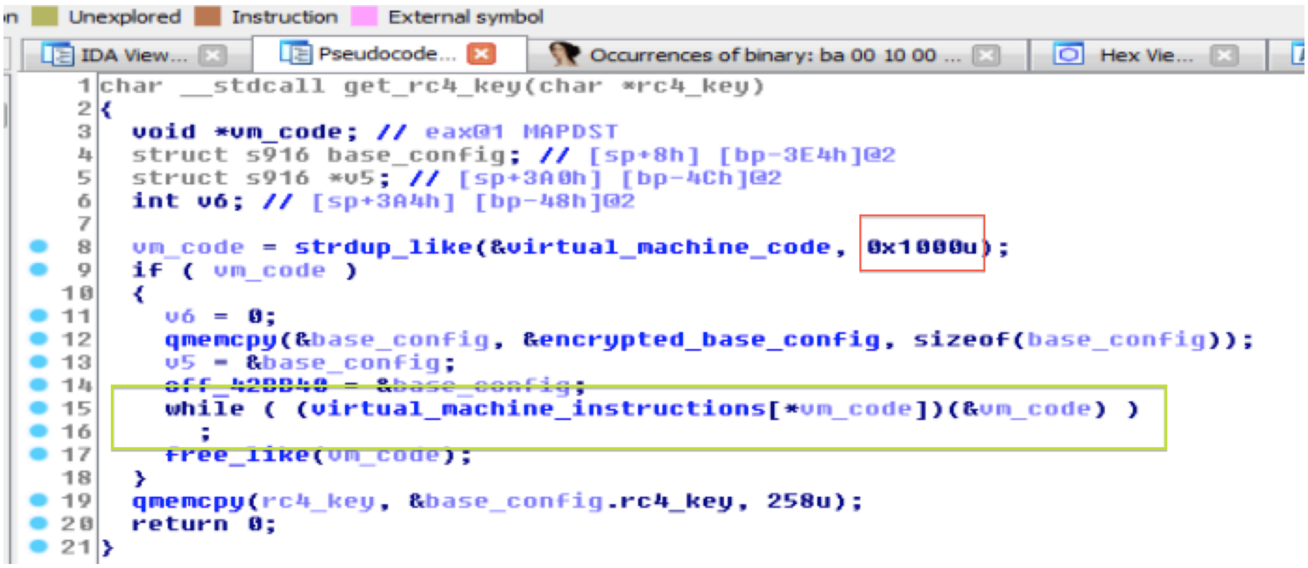
Once decrypted, the plaintext base config data structure can be parsed (see later section):

```
(Pdb) self.base_config
'\x16\xaf\x81\xe7\x13\xa08f\xec[{\x87\x0c\xab\xb5\x97\xe2Z]\xc0?\xff|}\xff\x17i
x18\\E\x9b\xd31\xf75<\x8fM\x1b\n\xe4\x18\x92\x00\x00L\xed\xb0!\x95\x11\x80>\xd3\xd
82+\x90\xf4\x05J\x80\x7f\x92U\x83C\x14\x9fsJ\xc2\x94\xddtDIM\xb4\x17X\xcc|2\xea\xa
c\x94~l\x94w+\xabE\\\xd7\xd0\x92w\xbc\xc3\x1b\xfe\x04\x8e\t\x81\xd4Y9z\x89\x99\x9e
e3p(\x17yw\xfe\x13C\x13v\t\x8e\xf3\xd1@L\xd24`X\xa6\x05?\x02\x1b\x98\xf7\xc7?\x0b<
\xfd\x9c\xb7\xb1O\x84\xa6*RnT"\xa3\x12\x08\x00\x04\x00\x87\x07\x00Xz\xc81\xaa\x12\
c\x92\xde\x91`*W\xa2\xfb/\x08\x93[\xb6\xbe\xcf\xd0 iS\xf0\xf9s\xb6\x8b Pf\x00\x00\
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
00\x00a\xd4\x89\xea8/\x13\xbc\xb8\x0e\x9d\xf1\xe0\x83\xbaavH\x10\xab\xf0\x0e](\x95
0\xbe>c6d\x97\xbaM\x10\x01p\x94E\x86\xb3\x8eG\xe9\x90)#\x94\x14L\xef\xd2\xb1\xafS>
xc5\xa3\xe5>\xc8\x9c7\xcc\xce\xb9\x9a\xbe\xeb\xcbd\x9f\xca\xe6+?Ls\x87ds\x81\x18\x
manager/csv/cache/file.php|file=sivtel.dll\x00\xc29\x91\xc2,\xe8\xfbb\xf7\xc6\xe4])
\xf4\x1f\xcf\x1aX\xaf\':\xc1Y\xb5\x04\x00\x01\x00\xf63\x8b\x07\xf56\xc4\xdaH\x1b\x
ec\xa4\x8a\\D\x9e\x17\xfb\x0f\x90w\x11C\xae\x18\x86\xc4J\xd1\x00\xcb\xe2#j\x9c\x8f
x16\x7f\xef\x161(\xda"6\x94?\x9dRc|\x98H`\x90b\xd9{S\xdd\x1c\xf5\xadt\x02\xb7fC\x9
\xae\xe3\xba\x87B\x80\x89\xb2\xa0W\xec?M\xba\x17\x8cD\xaf\xa3\xf6;J\xb4\x8b\x19\xc
da\xe8\xcd\xa1\xa1}\xbd\x15\xf3Q\x04\xb9g#z\xf2O\x0b\xfaF~\xf8\x9b:T\xe9\xbf\xdf\x
d0\xed\xf1;\x15\x1a\x1c\xdc\x89\xfe\xdc\xe2\xf7e\x02\\\xfa\xc9\x81\xa2\xff`g\xd3p\
xd1\xf4\xf4\xef{\x0f\x99Wx\xb6\x0f\xfc\xb9\xa3o\x9e\xcc\x00\x00\x00\x00\x00\x00\x0
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
F\x86/%\xe6g\x85}\t\xb6\xaf\xf8\xf0?\x9e\xd1\xb4E\xd4\x92\x89\xbd\xc8\xe4C\xba\x8c
\xa8\xc2\xc8(a\x9au\xc4\xe7</\xef\x92\xf2,\xdfV*.\x15>\x1d\xa0&\x15\xae\xe1\xc1f\x
\x1c\xd1\xa5~h\xf7UG\xbe%\x80\xef\x8f\x95\xc3\xaa\x119\x9b\x14?\xad\x85\xc5!kc%M>"
\xac\xd1&\xcb\x1f\xceoq\xecd\xf3V,\x1d\xa1\xfd`@6\x81MS\xb13\n\xdf\x07\x85Ws\x19\'
ea\xe1\x02\xbf\x97\x91\x11y\xf8\t\x08\x00\x00\x00\x11~]\x00B\'\xe7\x0be;|\x1b\x05!
03\xa6)\x0c\x14\x154"m{\xf5K\xeeD\xf3\x82'
```

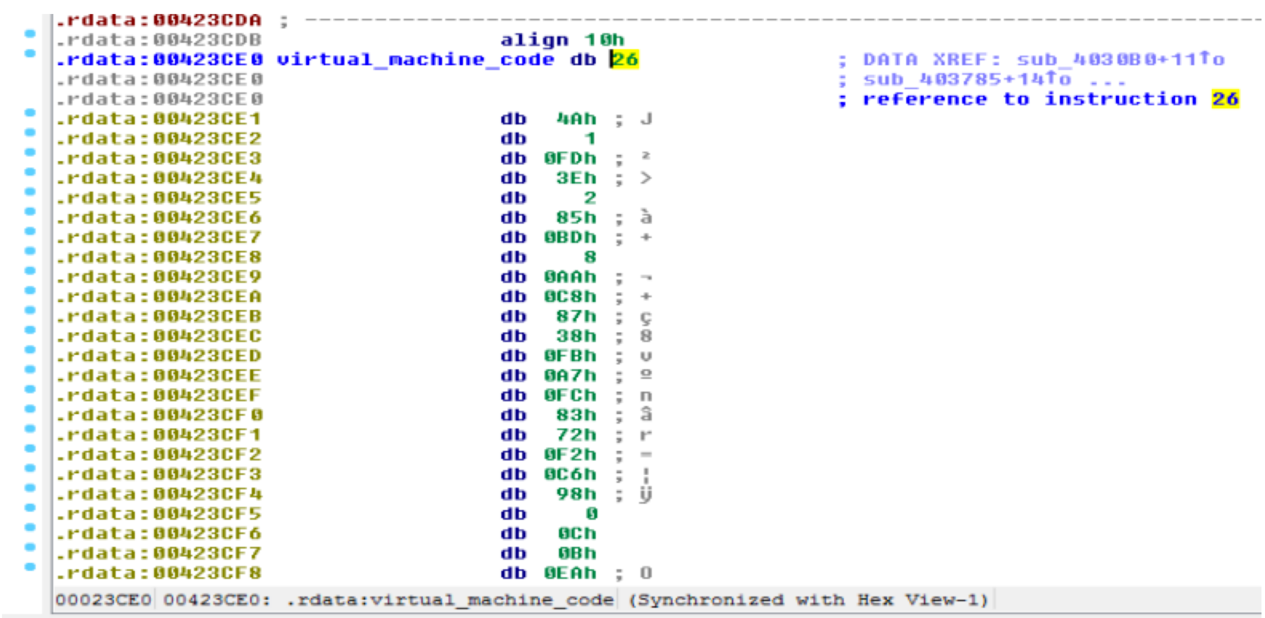

Decrypting the Base Config in ZeusVM

The most significant improvement in ZeusVM compared to other versions of Zeus is that instead of decrypting the base config with a simple XOR, it inputs the config to a custom virtual machine that runs and decrypts it. This is where the "VM" in ZeusVM comes from.

One way to track down the virtual machine components in the executable is by searching for PUSH or MOV instructions containing an immediate value of 0x1000 (highlighted in red). This usually returns a number of functions that contain code similar to:

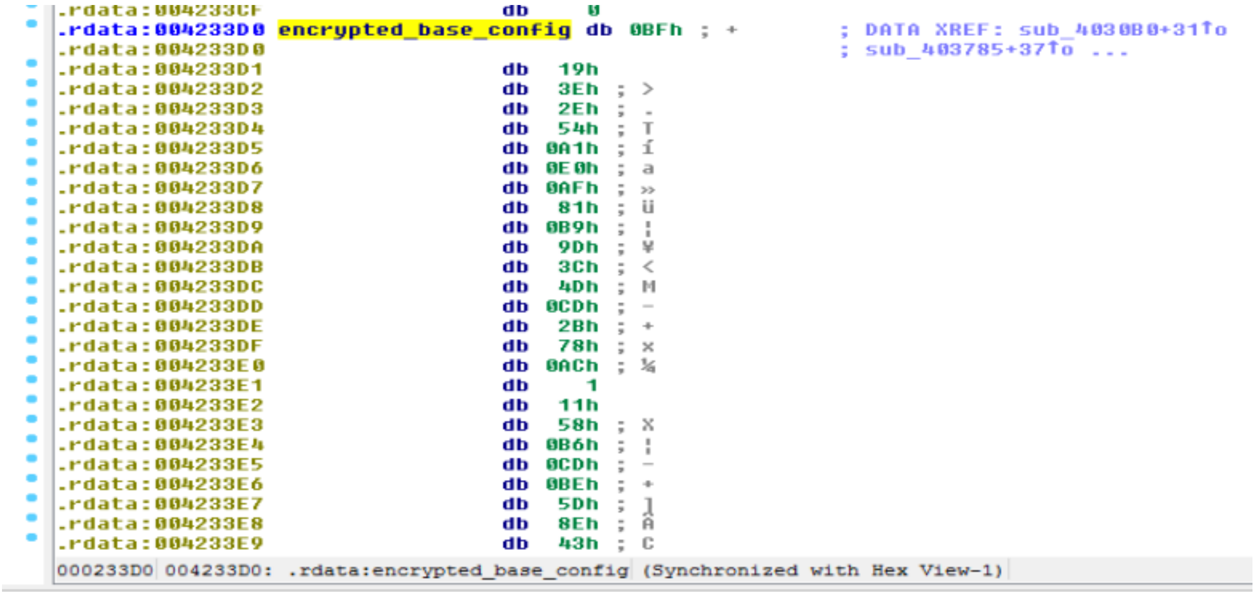


The virtual machine is composed of four components: code, data (encrypted base config), handler, and instructions. The first piece is the code and it is always 4096 (0x1000) bytes in size:

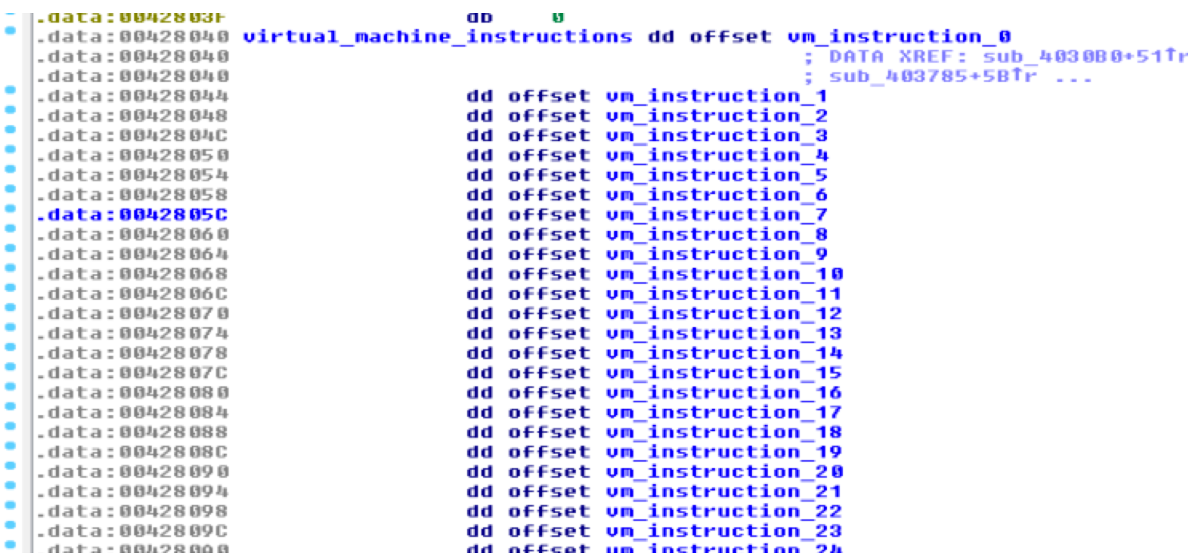


This opaque chunk of "code" consists of references to instructions and instruction data. For example, the first byte of the code, 26 (highlighted in yellow), is a reference to instruction #26.

The second piece, the data, is the encrypted base config. Its length is 916 bytes in this sample:



Next, the handler is implemented as a while loop that steps through the virtual machine code (highlighted in green in the above code screenshot). Each instruction reference (e.g. instruction #26 from above) is used as an index into an array of instructions:



The last component of the virtual machine are the instructions themselves. There are 69 instructions and they can be broken down roughly into the following groups:

- NOP (no operation)
- XOR
- Add
- Sub (subtract)
- ROL (rotate left)

- ROR (rotate right)
- Not
- Reorder
- RC4
- Set ECX
- Set EDX
- Loop
- Mov (constant value into register)
- Mov (data in registers)
- Add (data in registers)
- Sub (data in registers)
- XOR (data in registers)
- Add (constant value and register)
- Sub (constant value and register)
- XOR (constant value and register)
- Add (store result in memory)
- Sub (store result in memory)
- XOR (store result in memory)
- Mov (data in memory into register)
- Mov (data in memory)
- Leave (last instruction)

Instruction groups usually come in a set of three instructions each performing the same operation just on different data sizes. The supported data sizes are: byte, word (two bytes), and DWORD (four bytes). Most of the instructions perform basic arithmetic and logical operations. The most complicated instruction of the bunch is an implementation of the RC4 encryption algorithm.

An instruction can make use of 19 registers. 16 of them are general-purpose and the other three are used similarly to their x86 counterparts:

- EIP – instruction register
- EDX – data register
- ECX – counter register

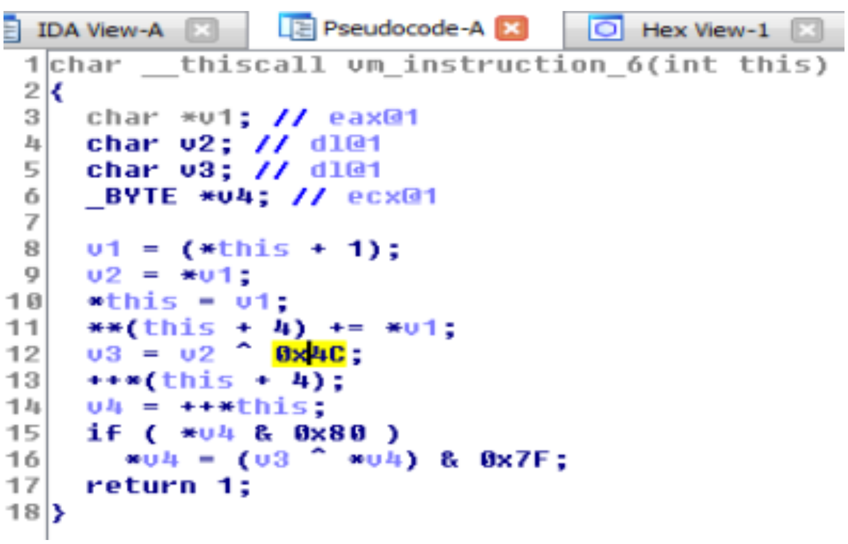
To get a general sense of the instructions, it is worthwhile to take a closer look at one of them. Instruction #6 performs an addition on two byte values. Using an offset from the instruction register, EIP, the first value is extracted from the virtual machine code. Likewise, using an offset from the data register, EDX, the second value is extracted from the virtual machine data. These two values are added together and the result is stored back in the data section replacing the prior value. The original source code implementing this instruction looks like this (highlighted in gray):

```
// ADD's, byte/word/dword sized
#ifdef BUILDER
#define instr_add(size) \
    static bool instr_add_##size(DEC_CONTEXT *ctx) {    \
        wsprintfA(szDbgMsg, "%X:"__FUNCTION__" edi=%X, ecx=%X; %X += %X (%X)\n", ctx->eip, ctx->edi - bEdiBase, ctx->ecx, *(size*)ctx->edi, *(size*)(ctx->ei
p + 1), *(size*)(ctx->eip + 1) + *(size*)ctx->edi);     \
        OutputDebugStringA(szDbgMsg); \
        ctx->eip++; \
        *(size*)ctx->edi += *(size*)ctx->eip;   \
        ctx->edi += sizeof(size);   \
        ctx->eip += sizeof(size);   \
        return true;    \
    }
#else
#define instr_add(size) \
    static bool instr_add_##size(DEC_CONTEXT *ctx) {    \
        BYTE bXorKey = ctx->eip[1] ^ magic_add_##size;  \
        ctx->eip++; \
        *(size*)ctx->edi += *(size*)ctx->eip;   \
        ctx->edi += sizeof(size);   \
        ctx->eip += sizeof(size);   \
        if(*ctx->eip & 0x80)    \
            *ctx->eip = (*ctx->eip ^ bXorKey) & 0x7F;   \
        return true;    \
    }
#endif

instr_add(BYTE);
instr_add(WORD);
instr_add(DWORD);
```

The rough outline of the code is:

1. Get sample-specific/instruction-specific XOR key (see below)
2. Perform the addition
3. Update the registers
4. Calculate the new EIP value using the XOR key

A #define macro is used so that it is easy to generate the instruction group using the three different data sizes. Another view of the same instruction is from IDA Pro:




```
char __thiscall vm_instruction_6(int this)
{
  char *v1; // eax@1
  char v2; // dl@1
  char v3; // dl@1
  _BYTE *v4; // ecx@1

  v1 = (*this + 1);
  v2 = *v1;
  *this = v1;
  **(this + 4) += *v1;
  v3 = v2 ^ 0x4C;
  ++*(this + 4);
  v4 = ++*this;
  if ( *v4 & 0x80 )
    *v4 = (v3 ^ *v4) & 0x7F;
  return 1;
}
```

This view highlights (in yellow) a sample-specific/instruction-specific XOR key that is used when updating the next EIP value. These XOR keys provide some randomness to the virtual machine. Besides these keys, the instructions have been the same from virtual machine to virtual machine. As another example, here is instruction #6 from the ZeusVM 2.0.14.0 sample with a different XOR key (highlighted in yellow):

```
char __thiscall vm_instruction_6(int this)
{
  _BYTE *v1; // edx@1
  char *v2; // eax@1
  char v3; // bl@1
  char v4; // bl@1
  _BYTE *v5; // ecx@1

  v1 = *(this + 4);
  v2 = (*this + 1);
  v3 = *v2;
  *this = v2;
  *v1 += *v2;
  v4 = v3 ^ 0xC9;
  ++*(this + 4);
  v5 = ++*this;
  if ( *v5 < 0 )
    *v5 = (v4 ^ *v5) & 0x7F;
  return 1;
}
```

For a final (and hopefully clearer) view of the instruction, here is a Python implementation:

```python
def op_6(self, key):
    """
    op_6 - instr_add_BYTE
    """
    xor_key = self.calc_xor_key(self.code[self.eip+1], key)

    # skip opcode byte
    self.eip += 1

    # edx
    arg = struct.unpack("B", str(self.code[self.eip:self.eip+1]))[0]
    data_arg = struct.unpack("B", str(self.data[self.edx:self.edx+1]))[0]
    val = struct.pack("B", (data_arg + arg) & 0xff)
    self.data = self.data[:self.edx] + val + self.data[self.edx+1:]

    self.edx += 1
    self.update_eip(1, xor_key)
    return True
```

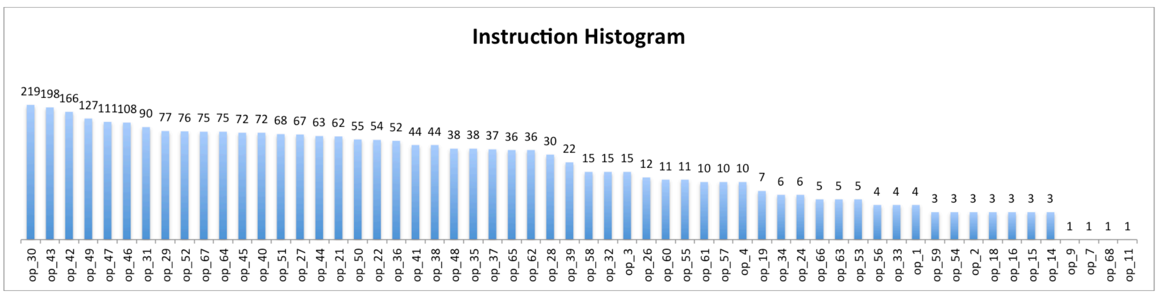
Putting the virtual machine components together and running it for this sample executes 2392 instructions. The first five are:

1. 26 – instr_setedx_SHORT
2. 26 – instr_setedx_SHORT
3. 22 – instr_rc4_crypt
4. 22 – instr_rc4_crypt
5. 26 – instr_setedx_SHORT

The last five are:

1. 52 – instr_xor_r_const_DWORD
2. 47 – instr_sub_r_const_BYTE
3. 36 – instr_add_r_r_WORD
4. 59 – instr_stos_xor_BYTE
5. 68 – instr_leave

A histogram of the run looks like:



The top five instructions are:

1. 30 – instr_mov_r_const_WORD
2. 43 – instr_xor_r_r_DWORD
3. 42 – instr_xor_r_r_WORD
4. 49 – instr_sub_r_const_DWORD
5. 47 – instr_sub_r_const_BYTE

And the bottom five instructions are:

1. 14 – instr_rol_DWORD
2. 9 – instr_sub_BYTE
3. 7 – instr_add_WORD
4. 68 – instr_leave
5. 11 – instr_sub_DWORD

Base Config Contents

Once decrypted, the base config looks like this:

```
(Pdb) self.base_config
'`\xabJ\x98\x19\x87\xdd\xae\xb1v/\x98\x8dU{=Ha\xd6\xe6XM\xfd\xf3\xef\x92Q\xcd \x07 \x1bC\xb1&9\x9a\xdcT\xe55j\xd0<\x12lhttp:
//olpfo.com/xapwj/cfg.bin\x00\xa3\xdc3\x1b\xcb\xf7m\x08n\x06\x92\xe7-\xfeO\xfd@\x01\xd0]\x06\xd9T.\x92\xdf>A\x07\xb8M\x85Mm\xf1\
xcd\xe8\x1dwOQ\xc23@1\x17[\x1c\x82\xf5n\':2\xd6K\xd4e\x18\xad\xc74\x10qCt\x0eyA\r\x94\xca\x7f\xea_\xdf\x83\x8f\x90\x05\\<\x9eRX\
xc1]W\x9b\xa2\xd7G/\x1e\xf8i\x08\xd3k\x12\x98J\xbb\x89\xcc\x81E\xc9\x8a\xab`%\xa4\x92= ^\x8cL\x07\x03\x0fb\xb4\x02\x85\xb2\xa6\x
19\xbf\xe9f.d\xb0\x86\xd8x\xff\tS6\xed\xeb&\x1fv\xbd\xfd\x1b\xd5\xdd\x9f\xa8\xfb\xc6\x9d}\x8eD\xc8\xb1\xfc\xef\xbe?\xb7\x1a\xb3U
\xda5\xdb\x99\xa9\xf7{#\xee\xe3!\xb6l\xf2\x9a\xa0\x15\xaa\x9c\x06\xc4\xac\x04h\xe0\x93\xcb\xe7BN\x84r\xd2I\xc0\x0bZ\xce\x01*\xa1
\xb8\xb9\xec\xe4\x88\xa3\x960\xe6\xd0\xf6\x0cH8(\x16\xbcj\xc5\x97\xf0\x14\xe2\xd9a\xdc,\xb5;\xf4\xcfY\xa5\xde\x87\xa7\xaeVo\x00~
\x91pu>$\xfas\xfe7\xf9\x11\x80Tz\nP\xf3\x959"\xe5)\xe1\xba\xd1\xaf\xc3\x8bg|F+-\x13\x8dc\x00\x00\xf9Vl\x14\x0f9\x98TH\xd7\x80\xe
c\xf9(h\xd0P\xa4\xc0s\xca\xf5}d\xd5\xf5PMXN(\xeb\xd7\x19\x8b\x02\xe7\x91p\xe8\xda\x00\xaf$\xe51\xfd\xe5|\xa7)S\xd7\xed\x02\x01&j
\x12\x9fe}\x01\xc58T\xe1\x85\xd9g\xa4\x92\xbfX\x98\x89\xf4n{\xb2f\xc7"\x1e\x0c\x0c?\x8e\x96\x0f=cU\xb9\x91]}m7\x0b\xd3p\x94\xe7\
xdd\xb4s \xc6\xa5\xedrR$\xfd\xb9\xc4K.\xdf\x171\xed\x8e\xa2\x80o\x804R[\xd7\xf8_\xca\x07uT\xa2\xdb\x7f.\x85\xaa\xeaM\xd4DW\xb8r\
xc39\xef<\tw\x98\xc5\xe5q\xec\xbb4\xf8b\x89Y\xfbL\xbe\xba\x9b\xa9\xaa\xf2\xb6*L\x82\x100eS\x9a\x11\xdcc\xc8\x96i\xac_\xc5C(\r\x0
3\x00\x03\x00\xbdR\x9a`\x1f\xdb2;\xe6\xc1\xfb\xe7\xb4l\xdb\xffA/\xdf\x9b\x02\\\xc6\xa8\t\xf0\x00Y1\x95p\x82N\xcfB\xa4\x01\xb05\x
8f\xdb\xbb\xd9\xc53\xa9\x02\xd93\x1e\x9c/\xb9\x8b\xb6l\x15L\xa7mM\xee\xc1\x1b&\xa2o(4\x15R1\xbb\x15\x11\xda\xcf1\x9e\xb0\x01-\x0
4\x1a\x071\x06\xeb\xbeq\xc9\xfc*\xb4u,*\x02\x1e^D\xa7E\x98\x82\xe1/\xa9!|\x9b\xdf\x19G\x9d\xae/\xbfD\xc8\xf5vs\x00p\x00r\x00i\x0
0n\x00g\x00\x00\x00;\x03\x80;h\xf4\x83\xefTV-\xc3\xa1R\xd2K\xe71f9UqA\x1c\x9c\x18\x1fG\x08Hk\x0e\xf0E\xcf,O\x19\x8dV<4#\xab/\xa1
FV\xd7\xdf\xb7\x03h\x01\xdc\x14]\xf8xx?\x90\xbb\xa71Rn\x15n\xd5\x9d\x12\x92:.bin\x00OP\x98Vq2)\xae$\xa2DD\xc8\xd3\xeb\x14hB\x81\
x07\x1e\xe1NJ\x01\x86\x80\x9b\x05U+@w\xfb\xb5\x17\xa6v\xb4\x88)\x8d\xf8g\xf4\x98wJ\xc3\xa6\x06\xe4\x8d&r\xe1\x89O\x16t\xac\xd8F\
xfa0aq\xfc^\xf9\x0b\x16\xfb\x17\xec\xb6H\xb5\xd2\xc0Y\xcc\xbe\xb9\x96K\xef\xfe\xed\xfb\x11F\x8f\x02\xe6\x85\xcd\x0c:\x93\xd9\x00
\x00\x08\x00\ts%\xeb\xfa\xdff\xdc\x03\x00\x03\x00\x03\x00\x03\x00=_\xca\xc6\x08\xcfBF\x13N%\xb8'
```


It has a number of useful items such as: bot name, fake command and control URL, real command and control URLs, and crypto "keys".

Bot Name

The bot name is an optional user configurable name. Per the example configuration file from the leaked builder the default (commented out) bot name is "btn1":

```
;Build time:   20:38:43 09.06.2014 GMT
;Version:      2.0.0.0

entry "StaticConfig"
  ;botnet "btn1"
  timer_session 1 1
  url_config "http://domain.com/folder/config.jpg"
  url_reserve_config "http://domain.com/folder/config.jpg"
  remove_certs 1
  disable_tcpserver 0
  encryption_key "put your key here"
end
```

When available, the bot name is stored using wide characters (highlighted in red in the above base config screenshot). For the two analyzed samples the names are: "spring" and "test". While bot names can be helpful when categorizing campaigns, they are not a unique indicator.

Fake/Decoy Command and Control URLs

When parsing a decrypted ZeusVM 2.0.0.0 base config for command and control URLs, one will always show up (highlighted in green in the above base config screenshot). This easily found URL is a fake/decoy meant to fool security researchers. Querying ASERT's 2.0.0.0 samples reveals 68 unique fake/decoys with the top five being:

1. hXXp://rqxba.com/cfg.bin (1309 samples)
2. hXXp://yvtvvibsp.com/ldpyd/cfg.bin (900 samples)
3. hXXp://urgalxjef.com/cfg.bin (685 samples)
4. hXXp://bzfdcp.com/cfg.bin (647 samples)
5. hXXp://byoziszt.com/cabpc/cfg.bin: (482 samples)

Version 2.0.14.0 does not contain this anti-analysis feature.

Real Command and Control URLs

As implied by the previous section, the real command and control URLs are hidden in the base config with another layer of encryption. Each of the URLs is RC4 encrypted (the next section will discuss the key) and the cipher text occupies 101 bytes of space. Specific offsets of where the URLs are stored within the config can be tracked down in the disassembly:

```
  }
EL_18:
  if ( (v32 & 0x7F000000) == 0x1000000 )
  {
    if ( (v32 & 0xFF0000) < 0x50000 )
    {
      v15 = 101;
      qmemcpy(&rc4_key, &base_config.rc4_key, 0x102u);
      rc4(&base_config.field_242, 101u, &rc4_key, 0);
      v17 = 1;
      v3 = &base_config.field_242;
    }
    else
    {
```

One way to locate these offsets is to find the RC4 decryption function and look for calls to it where the length is 101 bytes (highlighted in red). In this example URL, the offset (highlighted in yellow) from the start of the decrypted base config is 0x242 (578) bytes:

```
(Pdb) plaintext = self.rc4_keystate(self.rc4_key, self.base_config[0x242:0x242+101])

(Pdb) "".join(plaintext[:plaintext.find("\x00")])
'https://arrowtools.ru/xEZNzZEQuj8vJwsZ/flashplayer.jpg'
```

A quicker and easier brute force method of extracting the URLs is to try decrypting every 101 byte chunk of data starting from the beginning of the base config and checking for a URL in the plaintext.

Crypto "Keys"

There are two crypto "keys" stored in the base config. The first isn't actually a key, but a 258-byte output of the RC4 key-scheduling algorithm (KSA). While not always stored in the base config as such, the last two bytes of the key state, the index pointers: i and j, should be assumed to be zero. As discussed in the above section, RC4 is used to decrypt the command and control URLs. One way to find the offset for the RC4 key state is to first find the RC4 decryption function, then look for calls to it where the length is 101 bytes (highlighted in red). Next, look for a memcpy copying 258 bytes (highlighted in green) from the base config to a local variable:

```
if ( (v32 & 0xFF0000) < 0x50000 )
{
  v15 = 101;
  qmemcpy(&rc4_key, &base_config.field_6D, 258u);
  rc4(&base_config.c2_url1, 101u, &rc4_key, 0);
  v17 = 1;
  v3 = &base_config.c2_url1;
}
```

In this sample, the key state offset is 0x6d (109) bytes (highlighted in yellow) from the start of the base config:

```
(Pdb) self.base_config[0x6d:0x6d+258]
'Mm\xf1\xcd\xe8\x1dwOQ\xc23@1\x17[\x1c\x82\xf5n\':2\xd6K\xd4e\x18\xad\xc74\x10qCt\x0eyA\r\x94\xca\x7f\xea_\xdf\x83\x8f\x90\x05\\<\x9eRX\x
c1]W\x9b\xa2\xd7G/\x1e\xf8i\x08\xd3k\x12\x98J\xbb\x89\xcc\x81E\xc9\x8a\xab`%\xa4\x92= ^\x8cL\x07\x03\x0fb\xb4\x02\x85\xb2\xa6\x19\xbf\xe9f.d\
xb0\x86\xd8x\xff\tS6\xed\xeb&\x1fv\xbd\xfd\x1b\xd5\xdd\x9f\xa8\xfb\xc6\x9d}\x8eD\xc8\xb1\xfc\xef\xbe?\xb7\x1a\xb3U\xda5\xdb\x99\xa9\xf7{#\xee
\xe3!\xb6l\xf2\x9a\xa0\x15\xaa\x9c\x06\xc4\xac\x04h\xe0\x93\xcb\xe7BN\x84r\xd2I\xc0\x0bZ\xce\x01*\xa1\xb8\xb9\xec\xe4\x88\xa3\x960\xe6\xd0\xf
6\x0cH8(\x16\xbcj\xc5\x97\xf0\x14\xe2\xd9a\xdc,\xb5;\xf4\xcfY\xa5\xde\x87\xa7\xaeVo\x00~\x91pu>$\xfas\xfe7\xf9\x11\x80Tz\nP\xf3\x959"\xe5)\xe
1\xba\xd1\xaf\xc3\x8bg|F+-\x13\x8dc\x00\x00'
```

The second key isn't a key either, but a 176-byte RC6 key state/S-box. As will be discussed below, RC6 is used to decrypt the configuration file retrieved from the command and control server. The offset for this key can be found similarly to above: locate the RC6 decryption function and trace its key argument back to a memcpy of 176 bytes (highlighted in yellow) from the base config:

```
char __stdcall get_rc6_key(char *a1)
{
  _BYTE *v1; // eax@1
  struct s916 base_config; // [sp+8h] [bp-3E4h]@2
  _BYTE *v4; // [sp+39Ch] [bp-50h]@2
  struct s916 *v5; // [sp+3A0h] [bp-4Ch]@2
  int v6; // [sp+3A4h] [bp-48h]@2
  void *v7; // [sp+3E8h] [bp-4h]@1

  v1 = strdup_like(&virtual_machine_code, 0x1000u);
  v7 = v1;
  if ( v1 )
  {
    v6 = 0;
    qmemcpy(&base_config, &encrypted_base_config, sizeof(base_config));
    v4 = v1;
    v5 = &base_config;
    off_42BB40 = &base_config;
    while ( (virtual_machine_instructions[*v4])(&v4) )
      ;
    free_like(v7);
  }
  qmemcpy(a1, &base_config.field_178, 176u);
  return 0;
}
```

This sample has the RC6 key state offset at 0x178 (376) bytes (highlighted in red) from the start of the base config:

```
(Pdb) self.base_config[0x178:0x178+176]
'\xd7\x80\xec\xf9(h\xd0P\xa4\xc0s\xca\xf5}d\xd5\xf5PMXN(\xeb\xd7\x19\x8b\x02\xe7\x91p\xe8\xda\x00\xaf$\xe51\xfd\xe5|\xa7)S\xd7\xed\x02\x01&j\x12\x9fe}\x01\xc58T\xe1\x85\xd9g\xa4\x92\xbfX\x98\x89\xf4n{\xb2f\xc7"\x1e\x0c\x0c?\x8e\x96\x0f=cU\xb9\x91]}m7\x0b\xd3p\x94\xe7\xdd\xb4s \xc6\xa5\xedrR$\xfd\xb9\xc4K.\xdf\x171\xed\x8e\xa2\x80o\x804R[\xd7\xf8_\xca\x07uT\xa2\xdb\x7f.\x85\xaa\xeaM\xd4DW\xb8r\xc39\xef<\tw\x98\xc5\xe5q\xec\xbb4\xf8b\x89Y\xfbL\xbe\xba\x9b\xa9\xaa\xf2\xb6*L\x82\x100eS\x9a'
```

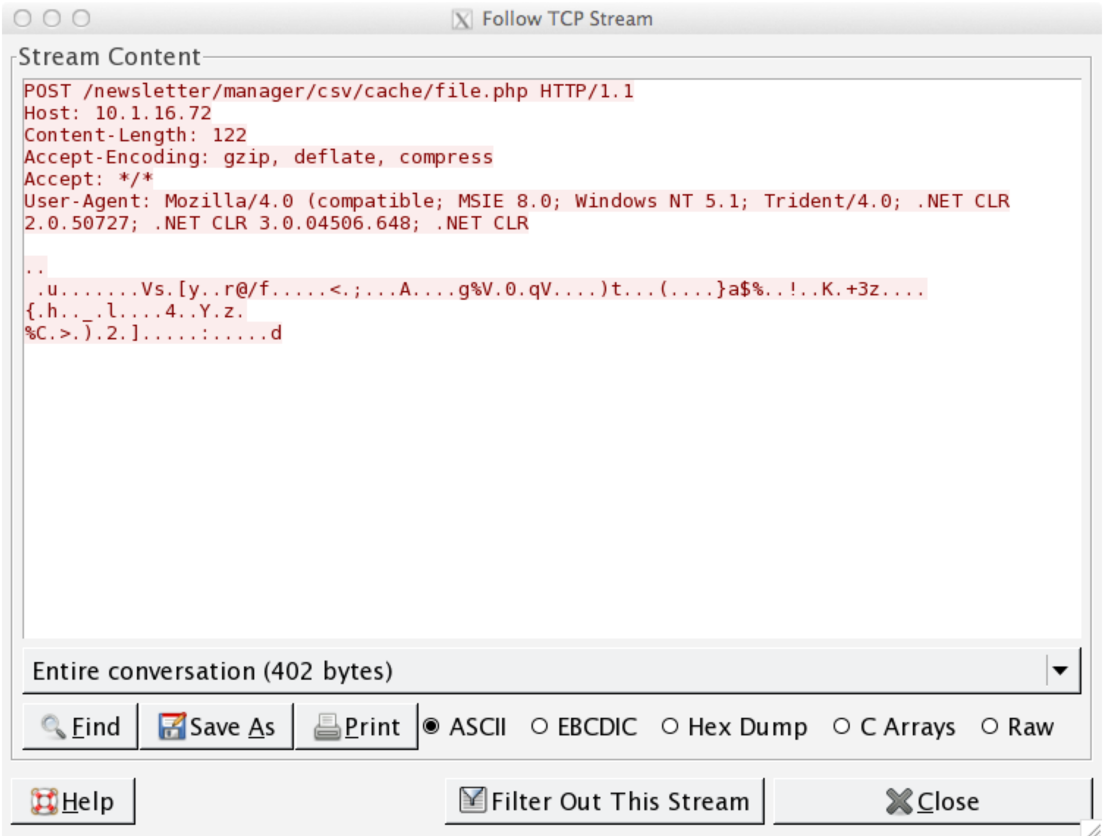
Configuration File

With the RC6 key and command and control URLs in hand, the next challenge is retrieving, decrypting, and parsing the configuration file from the command and control server. The most important elements from this config are the webinject rules used to control what data is stolen from which victim (more on this below).

Compared to its brethren, ZeusVM implements a radically different retrieval mechanism.

Retrieving the Command and Control Config in Other Zeus Variants

As before it is worthwhile to take an aside to review how other Zeus variants retrieve their configuration file from the command and control server. This example will use the same Citadel 1.3.5.1 sample as referenced above. Citadel's config request is an HTTP POST to the command and control URL with encrypted POST data:



The POST data is encrypted with two layers of encryption: modified RC4 and Zeus' visual encrypt. The first layer uses standard RC4, but it additionally XORs in the bytes of a 32 byte hardcoded "login key". The second layer called "visual encrypt" is a XOR based encryption that is common to Zeus variants. Decrypting the visual encrypt layer can be done with the following Python function:

```python
def visual_decrypt(self, message):
    """
    Zeus visual decrypt
    """
    plain = []

    for i in range(len(message)-1, 0, -1):
        plain_byte = ord(message[i]) ^ ord(message[i-1])
        plain.append(chr(plain_byte))

    plain.append(message[0])
    plain.reverse()

    return plain
```

Removing the layers of encryption reveals a binary data structure common to Zeus variants known as "binstorage":

```
    (Pdb) binstorage
    "W\xa0\xb3p\x9e\x9a>\xe2\x96\x98,A0\xf5\xb2\x18\x8d\x04g+z\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\xa9\xfc\x9d\x85\xa7*\
xd3tB_y\xeb:\r\xf1r%'\x00\x00\x00\x00\x00\x00 \x00\x00\x00 \x00\x00\x00C1F20D2340B519056A7D89B7DF4B0FFF&'\x00\x00\x00\x00\x00\x0
0\n\x00\x00\x00\n\x00\x00\x00sivtel.dll"
```

While the data types and content differ from variant-to-variant and command-to-command, the general format of the structure is:

- Header
    - Junk padding (20 bytes)
    - Size of section data (DWORD)
    - Flags/padding (DWORD)
    - Number of sections in section data (DWORD)
    - MD5 hash of section data (16 bytes)
- Section Data
    - Data type (DWORD)
    - Flags (DWORD)
    - Packed size (DWORD)
    - Unpacked size (DWORD)
    - Data
    - ...

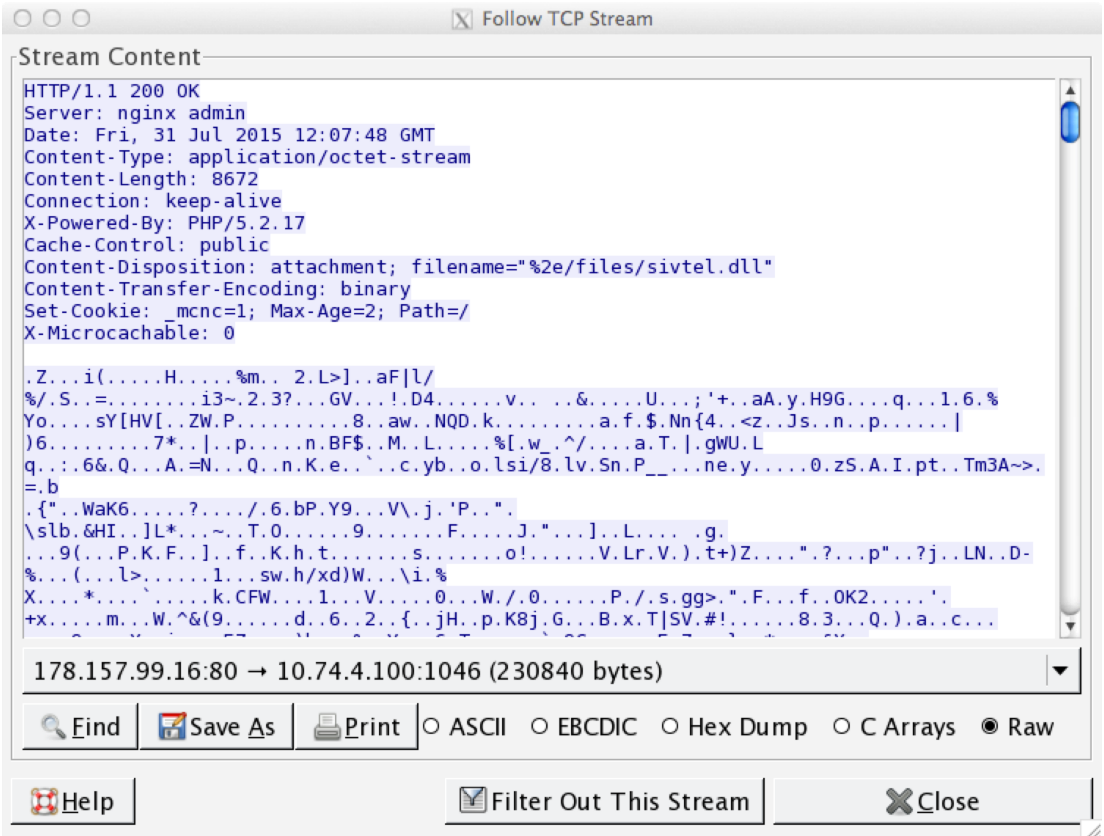
As described in Python, here's the header of the above request:

```
# header
# junk, hardcoded for ease
header = "57a0b3709e9a3ee296982c414ff5b2188d04672b".decode("hex")

# size
header += struct.pack("I", (len(section_0 + section_1) + 20 + 4 + 4 + 4 + 16))

# padding
header += struct.pack("I", 0)

# sections
header += struct.pack("I", 2)

# md5
md5 = hashlib.md5()
md5.update(section_0 + section_1)
header += md5.digest()
```

Next are the two sections that make up the section data:

```
# login key section
# type
section_0 = struct.pack("I", 10021)

section_0 += struct.pack("I", 0)

# size 1
section_0 += struct.pack("I", len(login_key))

# size 2
section_0 += struct.pack("I", len(login_key))

# data
section_0 += login_key

# config section
# type
section_1 = struct.pack("I", 10022)

# padding
section_1 += struct.pack("I", 0)

# size 1
section_1 += struct.pack("I", len(filename))

# size 2
section_1 += struct.pack("I", len(filename))

# data
section_1 += filename
```

In response to the POST, the Citadel command and control server returns the config:



The config is encrypted using three layers: AES, XOR, and visual encrypt. The first is a layer of AES-128 in ECB mode (using a generated AES key). Second is a basic XOR with the bytes of the previously mentioned login key and the last layer is visual decrypt.

Once decrypted, the config is presented in the binstorage format using config specific data types.

Retrieving the Command and Control Config in ZeusVM

As hinted by the command and control URL (hXXps://arrowtools.ru/xEZNzZEQuj8vJwsZ/flashplayer.jpg), this ZeusVM sample requests a .jpg over an HTTPS GET request. While not every sample uses TLS, every sample does request a .jpg file. The file is a legitimate JPEG that can be properly rendered:

But, as is expected, there is a wolf in sheep's (JPEG's?) clothing here. "A JPEG image consists of a sequence of segments, each beginning with a marker, each of which begins with a 0xFF byte followed by a byte indicating what kind of marker it is." [8] One of the markers (0xFF, 0xFE) indicates a text comment. Taking a closer look at flashplayer.jpg in hexdump reveals a JPEG comment (highlighted in red) that contains interesting looking data (highlighted in blue):

```
0005f0d0  4c f6 42 a8 67 33 d3 a7  ca 0c e7 2f 0e ba c3 ba  |L.B.g3...../....|
0005f0e0  01 2f de 92 69 f1 85 4d  e6 80 ea 37 a7 be 9a f6  |./..i..M...7....|
0005f0f0  c7 ff fe 3f 10 00 00 00  94 ec 4c 98 42 01 00 73  |...?......L.B..s
0005f100  48 47 61 30 6c 4a 46 4c  36 2f 36 4f 62 30 31 55  HGa0lJFL6/6Ob01U
0005f110  63 79 48 50 57 56 55 42  6e 74 41 7a 72 72 75 57  cyHPWVUBntAzrruW
0005f120  78 38 56 36 47 79 50 31  58 5a 38 74 65 66 37 38  x8V6GyP1XZ8tef78
0005f130  51 76 67 47 76 6b 62 6c  6b 44 63 65 35 66 4d 49  QvgGvkblkDce5fMI
0005f140  51 5a 64 72 7a 66 55 56  31 53 4a 6a 4e 6d 34 37  QZdrzfUV1SJjNm47
0005f150  2b 6f 63 47 77 6d 49 43  75 71 59 4f 6d 4c 38 35  +ocGwmICuqYOmL85
0005f160  4a 5a 52 48 79 6a 34 79  52 57 65 32 71 4b 43 54  JZRHyj4yRWe2qKCT
0005f170  74 6c 75 35 79 32 5a 77  44 65 63 45 49 2b 2b 62  tlu5y2ZwDecEI++b
0005f180  65 41 34 64 41 6a 76 39  69 6e 44 54 35 38 32 34  eA4dAjv9inDT5824
0005f190  51 57 51 6b 4a 45 38 33  78 61 64 58 35 62 48 47  QWQkJE83xadX5bHG
0005f1a0  4d 58 7a 5a 70 44 75 77  6b 6a 2b 72 44 47 4a 61  MXzZpDuwkj+rDGJa
0005f1b0  6c 42 73 77 79 4c 68 6a  68 50 59 6c 32 6a 76 6c  lBswyLhjhPYl2jvl
0005f1c0  67 73 2b 49 55 67 71 35  31 4a 50 67 32 45 4c 71  gs+IUgq51JPg2ELq
0005f1d0  7a 37 41 33 7a 42 68 4e  36 76 61 46 69 39 2f 4c  z7A3zBhN6vaFi9/L
0005f1e0  42 44 30 4e 62 50 76 54  6a 46 39 6c 35 59 6d 61  BD0NbPvTjF9l5Yma
0005f1f0  30 4b 51 32 42 69 33 71  4c 71 66 63 6a 4d 36 59  0KQ2Bi3qLqfcjM6Y
0005f200  36 75 68 70 63 48 4f 46  53 71 61 38 78 50 30 50  6uhpcHOFSqa8xP0P
0005f210  51 54 47 33 4c 43 34 4f  6a 70 55 68 31 36 64 6f  QTG3LC4OjpUh16do
0005f220  2f 50 58 41 49 42 4a 35  6e 53 32 50 35 7a 2b 61  /PXAIBJ5nS2P5z+a
0005f230  62 78 4f 30 42 6d 78 46  6b 44 51 43 49 39 45 4b  bxO0BmxFkDQCI9EK
```

This data starts 14 bytes from the comment marker and is encoded with base64. 10 bytes from the comment marker, is a DWORD that contains the size (highlighted in green) of the base64 chunk. It is 82584 bytes in this case, but in practice the comment is always at the end of the JPEG and is only followed by the 2 byte End of Image marker (0xFF, 0xD9).

Just to note: the leaked version 2.0.0.0 builder comes packaged with the following source JPEG:

ZeusVM 2.0.14.0 updates this basic steganography technique. The analyzed sample contains a number of command and control URLs:

- hXXp://sandvicaa.pw/kou/config3.jpg
- hXXp://lollipopp.pw/kou/config1.jpg
- hXXp://vassabgg.pw/kou/config2.jpg

The third URL returned this JPEG:

As previously discussed, this JPEG contains base64-encoded comments, but instead of just one, it contains multiple comments:

```
>>> [comment.start() for comment in re.finditer("\xff\xfe", jpeg)]
[72965, 138500]
```

Taking a closer look at the first one reveals some differences from version 2.0.0.0:

```
*
00011d00  00 28 a2 8a 00 ff fe ff  ff 30 31 30 33 46 00 2f  |.(.......0103F./
00011d10  57 6c 47 4b 4c 32 47 39  48 43 74 50 44 56 78 2b  WlGKL2G9HCtPDVx+|
00011d20  65 4b 75 55 6c 62 68 75  4d 4d 66 44 73 2b 37 75  eKuUlbhuMMfDs+7u|
00011d30  2f 5a 59 37 77 2f 35 43  59 54 68 49 67 36 68 58  /ZY7w/5CYThIg6hX|
00011d40  69 34 66 2f 6e 69 50 61  61 69 37 52 67 69 68 39  i4f/niPaai7Rgih9|
```

Here, the base64 data (highlighted in blue) starts at 10 bytes from the comment marker (highlighted in red). Next to the comment marker is a 2-byte size field (highlighted in green) indicating the size of the base64 data for this comment. The last thing to note is the "0103F" tag (highlighted in purple), which is likely used to distinguish ZeusVM JPEG comments from legitimate JPEG comments in the source image.

Each of these base64 comments are extracted and concatenated together in the same order as in the JPEG.

Decrypting the Command and Control Config in ZeusVM

In version 2.0.0.0 the configuration file is decrypted in three layers: base64, RC6, and visual decrypt. The RC6 layer uses the RC6 key state from the base config.

Version 2.0.14.0 decrypts the config in two layers: base64 and RC6. Visual decrypt has been removed.

As was the case for Citadel, the decrypted config is presented in the binstorage format using config-specific data types.

Parsing the Command and Control Config in ZeusVM

The first few bytes of the config binstorage data structure for this example looks like:

```
(Pdb) config[0:500]
'~\x96\xc1\xc6\xed\xf1\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00=\x00\x00\x00sa\x1d\x81\x00\x00\x00\x005N\x00\x00\x04\x00\x00\x00\
x04\x00\x00\x00\x04\x00\x00\x009\xf6/U\x00\x00\x00\x00"N\x00\x00\x04\x00\x00\x00!\x00\x00\x00!\x00\x00\x00http://icpiedimulera.i
t/flash.exe\x00\x00\x00\x00.N\x00\x00\x04\x00\x00\x00-\x00\x00\x00-\x00\x00\x00http://brokelowhi.com/flashplayer/mod_vnc.bin\x00
\x00\x00\x00#N\x00\x00\x04\x00\x00\x00/\x00\x00\x00/\x00\x00\x00https://arrowtools.ru/xEZNzZEQuj8vJwsZ/tree.php\x00\x00\x00\x00$
N\x00\x00\x05\x00\x00\x00i\x00\x00\x00\xa9\x00\x00\x00\xff\xff\xff\xffhttps://reybomerte.ru/xEZNzZEQuj\xfbm\xfb\xff8vJwsZ/flashp
\x04y!.jpg\x006\x05\xf6\xf6\x0fsuemno\nhot7\x01\xf6\xff.6unchangeclus8\x00\x00\x00p\x00\x00H\x00\x00\xff\x00\x00\x00\x00%N\x00\x
00\x05\x00\x00\x00\x8a\x00\x00\x00\xac\x00\x00\x00\xdb\xff\xff\xff!*.microsoft.com/*\x00!http:/\tm\xed\xff\xee\xfdyspace\x16\x15
*googleuser\x10\xd8\xb7\x0f\xfbntent\x18pip$skyp*\xf6\xff?l@odnoklassniki.ru\\?k\xbb\x03vkCakF\x16@*/l\xec\xc9no]in.\x88m'
```

It has several slight differences from Citadel, mostly in the binstorage header:

- Unknown 1 (0xc6c1967e) (DWORD)
- Size of section data (DWORD)
- Flags/padding (DWORD)
- Unknown 2 (0x0) (DWORD)
- Number of sections in section data (DWORD)
- Double SHA256 of section data (first four bytes of hash)
- Unknown 3 (0x811d6173) (DWORD)
- Unknown 4 (0x0) (DWORD)

Sections are composed of:

- Data type (DWORD)
- Flags (DWORD)
- Packed size (DWORD)
- Unpacked size (DWORD)
- Data (variable)
- Unknown (all sections except the last one) (DWORD)

The example config has 61 sections in its section data. To get a general understanding of the sections, it is worthwhile to analyze one in depth. Data type 20005 (also known as CFGID_HTTP_FILTER or WebFilters) specifies URLs and actions to take if the compromised machine browses to them. Actions to take include the following: don't log anything, log POST data, take a screenshot, or record a video. In the example, the packed size for this section is 138 bytes and the unpacked size is 172 bytes. The data for this section looks like:

```
(Pdb) data
'\xdb\xff\xff\xff!*.microsoft.com/*\x00!http:/\tm\xed\xff\xee\xfdyspace\x16\x15*googleuser\x10\xd8\xb7\x0f\xfbntent\x18pip$skyp*
\xf6\xff?l@odnoklassniki.ru\\?k\xbb\x03vkCakF\x16@*/l\xec\xc9no]in.\x88mp\x12atl\x10\x00\x00\x00\x06\x00\x00\t\x00\x00\xff'
```

While there are elements of plaintext in the data, it is actually compressed. Data compression is indicated by the 0x1 flag being set and if the unpacked size is larger than the packed size. The decompression algorithm used is UCL [9] and applying it to the compressed data results in:

```
(Pdb) data
'!*.microsoft.com/*\x00!http://*myspace.com*\x00!*googleusercontent.com*\x00!*pipe.skype.com*\x00!http://*odnoklassniki.ru/*\x00
!http://vkontakte.ru/*\x00@*/login.osmp.ru/*\x00@*/atl.osmp.ru/*\x00\x00'
```

Once all the binstorage sections are parsed, the config can be further cleaned up into something fairly human readable:

```
Prologue
================================
size: 61933 bytes
config flags: 0x00000000
# sections: 61
MD5: 73611d81

url_loader (20002)
================================
http://icpiedimulera.it/flash.exe

url_server (20003)
================================
https://arrowtools.ru/xEZNzZEQuj8vJwsZ/tree.php

AdvancedConfigs (20004)
================================
https://reybomerte.ru/xEZNzZEQuj8vJwsZ/flashplayer.jpg
https://suemnopshot.ru/xEZNzZEQuj8vJwsZ/flashplayer.jpg
http://unchangeclust.ru/xEZNzZEQuj8vJwsZ/flashplayer.jpg

WebFilters (20005)
================================
!*.microsoft.com/* (don't log)
!http://*myspace.com* (don't log)
!*googleusercontent.com* (don't log)
!*pipe.skype.com* (don't log)
!http://*odnoklassniki.ru/* (don't log)
!http://vkontakte.ru/* (don't log)
@*/login.osmp.ru/* (screenshot)
```

While this is only the first part of a config (see external Appendix 1 for the full version), it starts to give an idea of some of the sections (e.g. WebFilters as explained above) and what data they contain.

Webinjects

The most important sections in the configuration file retrieved from the command and control server are the webinjects. In conjunction with Zeus' man-in-the-browser (MITB) functionality [10], webinjects specify "what to steal from where" from a compromised machine. It essentially lets the attacker have complete control over websites (such as a bank) that are being visited (regardless whether it's over TLS) by a victim.

All Zeus webinjects follow the same format. Here is one of the default webinjects distributed with the leaked builder that shows the basic outline:

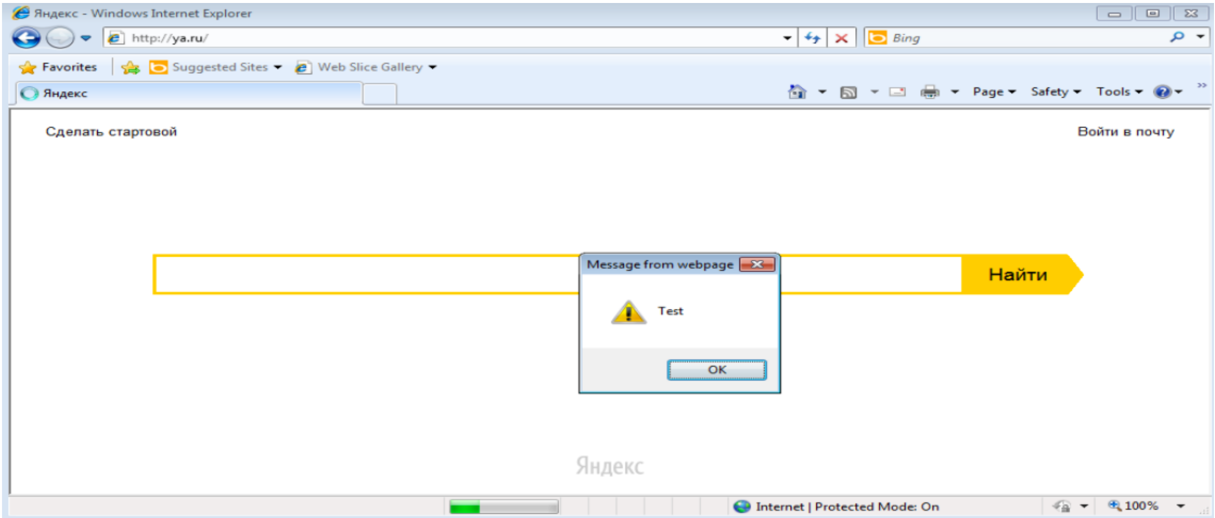
```
set_url http://ya.ru* GP
data_before
</body>
data_end

data_inject
<script type="text/javascript">
alert("Test")
</script>
data_end

data_after
data_end
```

The target URL is defined in the "set_url". It supports basic wildcarding and a few flags such as filter on (G)ET or (P)OST requests.

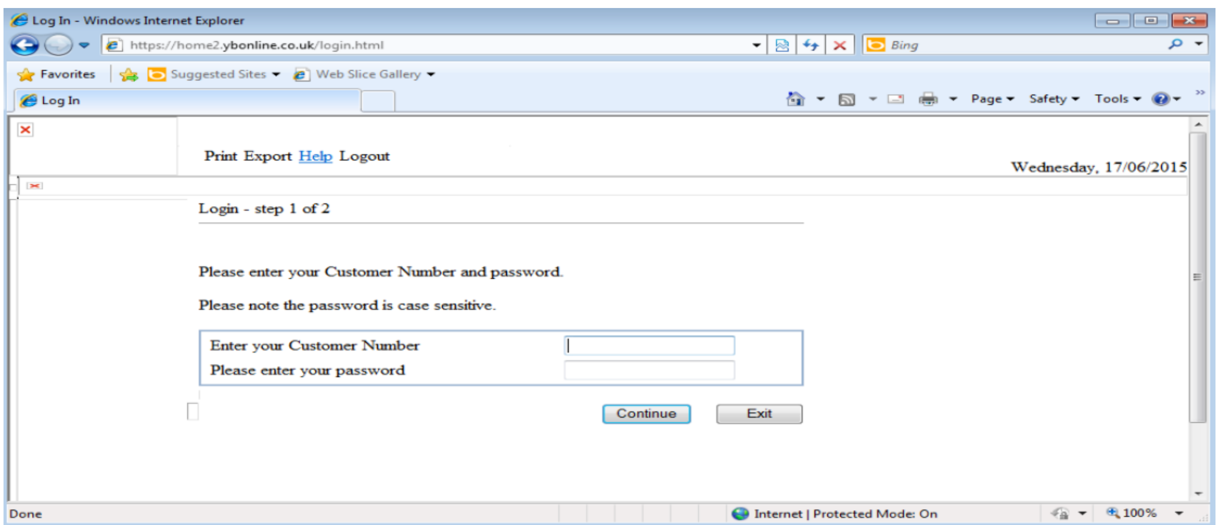
"data_inject" specifies the malicious code/data that the attacker wants injected into the targeted website. There is also support for basic substitution macros.

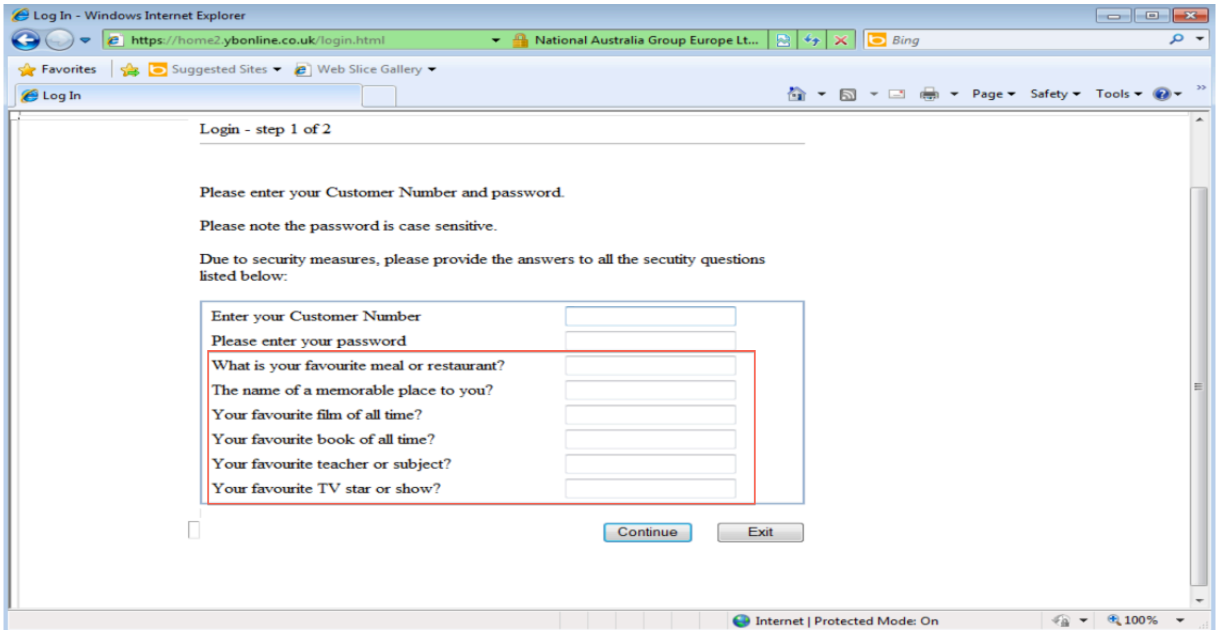
"data_before" and "data_after" control the position within the target's website source code where the malicious code/data should be injected.

This example injects some simple JavaScript that pops up a "Test" alert box:



As an example of a more malicious webinject, here's the before and after of a webinject targeting an HTTPS login site that adds fake form fields to social engineer additional information from the victim:





The contents of these extra, fake form fields are captured by Zeus' man-in-the-browser mechanism and sent to the command and control server. These basic (and old) webinject examples just scratch the surface of what webinjects are capable of, but should be enough to give a general understanding of the tactic.

Conclusion

This paper has taken a look at bits and pieces of the ZeusVM "banker" from a reversing perspective. Specifically:

- Versioning
- Decryption of the base config via custom virtual machine
- Interpretation of the base config including identification of bot name, decoy command and control URLs, real command and control URLs, and crypto "keys"
- Retrieval of the command and control config via JPEG files
- Decryption and parsing of the command and control config
- Basic webinject analysis

This should help organizations better understand, detect, and protect against this ongoing threat.

Components that were not discussed in detail were:

- Man-in-the-browser (MITB) implementation. This consists of process injection, function hooking, webinject parsing, injection, and data capture [10]

- Data exfiltration mechanism. This is done via an HTTP POST to the command and control server with an encrypted binstorage payload
- Non-webinject sections of the command and control configuration file. A number of these sections can be derived by cross-referencing data types to CFGIDs [12] and studying sample data

References

[1] https://blogs.rsa.com/is-cybercrime-ready-to-crown-a-new-kins-inth3wild/
[2] http://blog.fox-it.com/2013/07/25/analysis-of-the-kins-malware/
[3] http://securityblog.s21sec.com/2013/11/zeus-timeline-i.html
[4] https://github.com/Visgean/Zeus
[5] https://github.com/nyx0/KINS
[6] http://securityintelligence.com/zeus-maple-variant-targets-canadian-online-banking-customers/#.Vb-QvOsVTmw
[7] http://blog.malwaremustdie.org/2015/07/mmd-0036-2015-kins-or-zeusvm-v2000.html
[8] https://en.wikipedia.org/wiki/JPEG
[9] http://www.oberhumer.com/opensource/ucl/
[10] https://asert.arbornetworks.com/citadels-man-in-the-firefox-an-implementation-walk-through/
[11] https://asert.arbornetworks.com/neverquest-a-global-threat-targeting-financials/
[12] https://github.com/nyx0/KINS/blob/master/source/common/config.h