An introduction to Computer Virology

Jean-Yves Marion
LORIA
Université de Nancy

Jean-Yves.Marion@loria.fr
Some great stories !?

Waledac

C&C SERVER

PROTECTORS

REPEATERS

SPammers

Stuxnet

The Vast Reach of ‘GhostNet’
Researchers have detected an intelligence gathering operation involving at least 1,295 compromised computers. Below, the locations of 347 of the compromised machines, many of which were tracked to diplomatic and economic government offices of South and Southeast Asian countries.

GhostNet

Circles are scaled in proportion to the number of compromised computers found in each country.

Source: Information Warfare Monitor

THE NEW YORK TIMES

mardi 20 décembre 2011
What is a malware?

- A malware is a program which has malicious intentions
- A malware is a virus, a worm, a spyware, a botnet ...
- Giving a mathematical definition is difficult

🌟 So how it works?
How do infections by malware work?

You can’t patch stupidity

Social engineering

Infections

Mutations

Vulnerabilities

Bugs are un-avoidable
A case Study : Waledac

• Waledac is a botnet

• The goal is to send spams
Social engineering

WALEDAC recently used the Christmas 2008 holidays for its social engineering ploy when we first spotted it. It has changed its social engineering tactic of spamming on holidays and in relation to current economic events seven times. Appendix A chronicles the social engineering techniques we have seen this botnet use throughout its lifetime.

In addition to tricking users to run malware on their computers, WALEDAC also consistently populates inboxes with pharmaceutical (pharma) spam—spam that advertise Viagra, Cialis, and other similar sexual-enhancement drugs. However, there are times when WALEDAC spews out spam that are neither pharmaceutical in nature nor carry other malware. This suggests that it may have been hired by third parties or clients as a spamming service. These regular WALEDAC spam are also documented in detail in Appendix B.

The timeline shown in Figure 2 summarizes the WALEDAC activities seen so far.

Figure 2. Timeline of WALEDAC activities

At the time this report was written, WALEDAC was seen to have used eight social engineering attacks in an effort to make would-be victims run the malware. WALEDAC started out with the Christmas Ecard ploy.

Figure 1. Christmas ecard spam

Figure 2. Christmas ecard website

With the U.S. presidential election flurry coming to a crescendo in January 2009, WALEDAC started sending out spam for its new campaign. The email campaign then carried the bad news that “Obama refused to be the next president.”

Figure 5. WALEDAC email carrying the news that Obama refuses to be the next U.S. president

Figure 6. WALEDAC rips text off from Obama’s website, bearing false news that he no longer wants to be the president

After taking advantage of Obama’s presidential campaign, WALEDAC then turned its sights to Valentine’s Day.

Figure 3. Valentine's day website

Figure 4. Valentine's day ecard spam

2009-01-09
- Email spam changed tack to New Year
- Still the same eCard website

2008-12-26
- WALEDAC first seen
- eCard theme, in time for the Christmas season

2009-01-18
- Changed social engineering tack to “Obama refused to be a president”

2009-02-10
- Still using the Valentines theme, but now has the additional text: “Just in case you haven’t noticed yet…”

2009-02-17
- Still using the Valentines theme, but is now using several pictures in rotation.

2009-02-23
- Changed to “Coupons” theme, taking advantage of the current financial crisis

2009-02-28
- Websites added GeoIP component.
- Still using the Coupons theme, but the websites now know from what city you connected from.

2009-03-16
- Changed to “Bombing” theme
- Websites leveraged GeoIP for more targeted and localized social engineering

2009-04-15
- Engages users to spy on somebody else’s SMS messages.
Waledac

• Use of a dropper
  
  • Confickers or other malware are used to install waledac

• Binaries are packed with UPX and homemade packers

• Encryption technologies : RSA & AES (openSSL)

• Send emails from templates

• Scan files to find email addresses and password

• Download binaries and update itself
An email template

Received: from %C0%P%R3-6%
 %:qwertyuiopasdfghjklzxcvbnm%^%
 ([%C6%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.%^I%^%.^
Packer codes

Two different versions obtained after a few hours
3-tier movie of an attack

- Social engineering
- Targeted attack
- Client-side exploit
- Install malware
- A dropper
Software bugs
Software Bugs (client side exploit)

- Data are programs

- Bugs are doors if there are exploitable (0-days)

- A no bug system is safe

- But systems contain bugs ... and bug-free programs do not exist as a consequence of the undecidability of the halting problem (Turing)
Vulnerabilities : a buffer-overflow

```c
void vulnerable(char *user_data) {
    char buffer[4];
    strcpy(buffer, user_data);
}
```

![Stack diagram with buffer overflow]

---

mardi 20 décembre 2011
Vulnerabilities : a buffer-overflow

```c
void vulnerable(char *user_data) {
    char buffer[4];
    strcpy(buffer, user_data);
}
```

return at the address FFF0

![Stack Diagram]

- EIP: \x00\xFF
- EBP: \x00\xE0
- Buffer: \x00\x90
- Stack: \x00\x90...
Bugs

- A buffer-overflow transforms a program into a self-modifying program.

Wave 1 → Wave 1

Wave 2

Wave 2 is the code created by buffer overflow in wave 1.
What is a malware?

- Infect systems by **self-replication**
  - Mutation
- Protect itself
  - Obfuscation
  - **Self-modification**
- Detection
  - Undecidable
Outline

• Foundation 1 : Self-replication

• Foundation 2 : Self-modification

• Detection
  • Detection by string matching
  • Behavioral detection
  • Botnet neutralization
Foundation 1: Self-Replication
Self-replicating Cellular automaton

Von Neumann (1952), Burke

Codd, Langton
Cohen’s formalization (1985)

Consider Turing Machine M

and a Viral set $V$

When a TM $M$ reads $v \in V$, $M$ produces $v' \in V$

$(M, V)$ is a description of a virus
Self-Replication

- A virus has self-replicating capacity
- Reflexive property of programming language based on a
  - Pointer mechanism to program code, e.g. $0$

```bash
for FName in *; do
  # if FName is not me
  if [ $FName != $0 ]; then
    # add myself at the end of FName
    cat $0 >> $FName
  fi
done
```
Self-Replication

- Program encoding
  (Ken Thompson «Reflections on trusting Trust», CACM84)

```c
main() { char *s="main() { char *s=%c%s %c;
printf(s,34,s,34); }"; printf(s,34,s,34);}
```

- See also quines
Self-Replication

- Fixed point combinators (functional programming languages)

\[ YF = Y(YF) \]

- Computability: Recursion theorem of Kleene (1938)
  - See book of N.D Jones for a programming point of view
  - See books of Rogers or P. Odifreddi for a computability point of view
  - See marvelous books of R. Smullyan ...
A compilation point of view

A worm X scenario

- Open an email attachment by social engineering
- X scans for informations
- X extracts a list of email address of targeted peoples
- X sends copy of itself by email
Worm X specification

WormX(v,out)
{ info := extract(out); send(«badguy»,info); @bk := findAddress(out); send(@bk,v); }

How to compile Worm X ?
Program Semantics

$$\sem{P}(d)$$ is the value of $P$ on the environment $d$

$$\sem{-} : \text{Programs} \times \mathcal{D}^* \rightarrow \mathcal{D}^*$$

where a value of $\mathcal{D}^*$ is a system environment.
Solve a fixed point equation

We have to solve a fixed point equation:

\[ [W](out) = [\text{WormX}](W, out) \]

W is a worm satisfying the specification WormX
Kleene’s recursion theorem

If \( p \) is a program, then there is a program \( e \) such that:

\[
[e](x) = [p](e, x)
\]

Proof:

\[
[d(x)](y) = [x](x, y)
\]

\[
[q](y, x) = [p](d(y), x)
\]

\( e = d(q) \)

\[
[d(q)](x) = [q](q, x)
\]

\[
[d](q) = [p](d(q), x)
\]
Malware construction from a specification

Kleene recursion theorem:
If $p$ is a program, then there is a program $e$ such that:

$$\llbracket e \rrbracket(x) = \llbracket p \rrbracket(e, x)$$

Kleene fixed point is a solution of

$$\llbracket W \rrbracket(out) = \llbracket \text{WormX} \rrbracket(W, out)$$

We can construct a virus, which satisfies a given specification.
Self-replicating malware compiler

There is a compiler $\text{Comp}$ such that $W$ is the worm satisfying the specification $\text{Worm}$:

$$\left[\text{Comp}\right](\text{Worm}) = W$$
$$\left[W\right](\text{out}) = \left[\text{Worm}\right](W, \text{out})$$
Self-replication with mutations

We can generate malware which satisfies the specification *Worm*, and mutates at each execution automatically

\[ [e](out) = [\text{Worm}](\text{Mutate}(e), out) \]

where Mutate is a code mutation procedure

The construction of e is given by Kleene’s theorem.
Self-replicating compiler with mutations:

There is a compiler $\text{Comp}$ such that for all worm specification $\text{Worm}$ and mutation procedure $\text{Mutate}$:

$$\llbracket\text{Comp}\rrbracket(\text{Worm}) = \text{W}$$

$$\llbracket\text{W}\rrbracket(\text{out}) = \llbracket\text{Worm}\rrbracket(\text{Mutate}(\text{W}), \text{out})$$
References

• PhD thesis of F. Cohen
• L. Adleman (1988) which coins the word «virus»

A Virus is a Virus, Lwoff
Foundation 2 : Self-modifications
A data is a code

Today’s computers are built from two key principles:

1) Instructions are represented as numbers

2) Programs can be stored in memory to be read or written just as numbers

(Patterson & Hennessy)
Applications of self-modifying programs

- Malware mutations
- Code protection (digital rights)
- Compression and packers
- Just in Time compilers
A simple self-modifications

A simple decryption loop

@a: mov esi, $index
@b: xor [@offset + esi], $key
@c: sub esi, 4
@d: jnz @b
@offset: [encrypted data]

Wave 1

{a,b,c,d}

jnz @b

Wave 2

{offset}
Ex: Packer UPX

UPX (Packer)
Hello.exe

Wave 1

Wave 2
Another example of self-modification

Proxy =
{ X:= Read();
  eval(X);};

An external input is run

An interpreter of a known or unknown language is used to execute some data
Analyzing self-modifying programs

• Complex to design and to analyze

• Program flow may change

• Usual in semantics program and data are separated

Define by structural induction on \( P \): \( P \vdash \sigma \rightarrow \sigma' \)
Dynamic analysis of self-modifying programs

- Instrument a program
- Monitor read R, write W memory access and memory execution X
  - We follow nested self-modifying
  - We detect some code protection
  - We detect code patterns
    - code decryption
    - Integrity checking
    - ...

*Figure: Yoda Protector*
AC Protect
Themida

Wave 1
48709 ins.

Wave 2
394 ins.

Wave 3
5790 ins.

Wave 4
2628 ins.

Wave 5
2135 ins.

Wave 6
107078 ins.

Wave 7
2017 ins.

Wave 8
332 ins.

Wave 9
125 ins.

Wave 10
3994 ins.

Wave 11
30 ins.
Experiments with TraceSufer

TraceSufer based on Pin (Intel)

95 613 Binaries, 80% of success, 1400 binaries/h

Number of waves detected - max=56
Related works

- TraceSufer based on PIN (INTEL)
- Bitblaze (Berkeley) : TEMU, VINE, ...
- DynamoRio, Ether, Metasm
- Data tainting related methods
Malware detection
Malware detection by string scanning

• Signature is a regular expression denoting a sequence of bytes

Worm.Y
Your mac is now under our control!

• Signature: «Your * is now under our control

Worm.Y
Your PC is now under our control!
Malware detection by string scanning

- Signature is a regular expression denoting a sequence of bytes
- Data base of known signatures

<table>
<thead>
<tr>
<th>Produit</th>
<th>Taille signature (en octets)</th>
<th>Signature (indices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avast</td>
<td>8</td>
<td>12,916 → 12,919 12,937 → 12,940</td>
</tr>
<tr>
<td>AVG</td>
<td>14,575</td>
<td>533 → 536 - 538 - ...</td>
</tr>
<tr>
<td>Bit Defender</td>
<td>8,330</td>
<td>0 - 1 - 60 - 128 - 129 - 134 - ...</td>
</tr>
<tr>
<td>Dr Web</td>
<td>6,169</td>
<td>0 - 1 - 60 - 128 - 129 - 134 - ...</td>
</tr>
<tr>
<td>eTrust/Vet</td>
<td>1,284</td>
<td>0 - 1 - 60 - 128 - 129 - 134 - ...</td>
</tr>
<tr>
<td>eTrust/Inoculate IT</td>
<td>1,284</td>
<td>0 - 1 - 60 - 128 - 129 - 134 - ...</td>
</tr>
<tr>
<td>F-Secure 2005</td>
<td>59</td>
<td>0 - 1 - 60 - 128 - 129 - 546 - ...</td>
</tr>
<tr>
<td>G-Data</td>
<td>54</td>
<td>0 - 1 - 60 - 128 - 129 - 546 - ...</td>
</tr>
<tr>
<td>KAV Pro</td>
<td>59</td>
<td>Identique à celle de F-Secure 2005</td>
</tr>
<tr>
<td>McAfee 2006</td>
<td>12,127</td>
<td>0 - 1 - 60 - 128 - 129 - 134 - ...</td>
</tr>
<tr>
<td>NOD 32</td>
<td>21,849</td>
<td>0 - 1 - 60 - 128 - 129 - 132 - 133 - ...</td>
</tr>
<tr>
<td>Norton 2005</td>
<td>6</td>
<td>0 - 1 - 60 - 128 - 129 - 134</td>
</tr>
<tr>
<td>Panda Tit. 2006</td>
<td>7,579</td>
<td>0 - 1 - 60 - 134 - 148 - 182 - 209...</td>
</tr>
<tr>
<td>Sophos</td>
<td>8,436</td>
<td>0 - 1 - 60 - 128 - 129 - 134 - 148...</td>
</tr>
<tr>
<td>Trend Office Scan</td>
<td>88</td>
<td>0 - 1 - 60 - 128 - 129 - ...</td>
</tr>
</tbody>
</table>

Worm.Bagle.P:

Source: Filiol
Malware detection by string scanning

**Pros:**
- Accuracy: low rate of false positive
  - programs which are not malware are not detected
- Efficient: Fast string matching algorithm
  - Karp & Rabin, Knuth, Morris & Pratt, Boyer & Moore

**Cons:**
- Signature are quasi-manually constructed
- Vulnerable to malware protections
  - Mutations
  - Code obfuscations
Detection by integrity check

- Identify a file using a hash function

**Files**  **Hash function**  **Hash numbers**

**Cons:**
- File systems are updated, so numerical fingerprints change
- Difficult to main in practice
- Files may change with the same numerical fingerprint (due to hash fct)
Behavioral analysis

• Monitor program interactions (sys calls, network calls, ...)

• Detection of program behavior from execution traces

• Functionalities are expressed at high level

• Information leak can be detected

• **Cons**:  
  • Today behavioral analysis is dynamic, which implies to monitor all processes, or run programs in sandboxes.  
  • It slows down the system
Code protection

Detection is hard because malware are protected

1. Obfuscation

2. Cryptography

3. Self-modification

4. Anti-analysis tricks
Protections: Self-Modification and Obfuscation

• A lot of malware families use home-made obfuscations, like packers to protect their binaries, following a standard model.

• The obfuscation mechanism is automatically modified for each new distributed binary.

• For a human analyst, it is very hard to understand an obfuscated code
Win32.Swizzor Packer
Protections: Self-Modification and Obfuscation

• A lot of malware families use home-made obfuscations, like packers to protect their binaries, following a standard model.

• The obfuscation mechanism is automatically modified for each new distributed binary.

• For a human analyst, it is very hard to understand an obfuscated code because not all the code lines are meaningful and because x86 semantics is very tricky.

• One problem is the absence of high level abstraction to structure and understand obfuscated codes.
Code protection

Detection is hard because malware are protected.

Some interesting protection methods:

1. Obfuscation

2. Cryptography

3. Self-modification

4. Anti-analysis tricks
Obfuscation

- Function calls

- A function has a purpose

- Divide and conquer approach by understanding function purposes

- Higher degree of abstraction

- There are a lot, a lot of other obfuscation methods ...
Obfuscation

- but in malware’s world, what is the purpose of a function call?

Win32.Swizzor’s packer
Code protection

1. Obfuscation

2. Cryptography

3. Self-modification

4. Anti-analysis tricks
Cryptography

• The dropper of Agobot botnet
public static byte[] decrypt(byte[] message, string password)
{
    byte[] bytes = Encoding.UTF8.GetBytes(password);
    int num = message[message.Length - 1] ^ 0x70;
    byte[] buffer2 = new byte[message.Length + 1];
    int num4 = message.Length - 1;
    for (int i = 0; i <= num4; i++)
    {
        int num2;
        buffer2[i] = (byte)((message[i] ^ num) ^ bytes[num2]);
        if (num2 == (password.Length - 1))
        {
            num2 = 0;
        }
        else
        {
            num2++;
        }
    }
    return (byte[])Utils.CopyArray((Array)buffer2, new byte[(message.Length - 2) + 1]);
}

http://quequero.org/AgoBot_Botnet_Reverse_Engineering
Code protection

1. Obfuscation

2. Cryptography

3. Self-modification

4. Anti-analysis tricks
Self-modification

- Packers ... already seen previously ...
Code protection

1. Obfuscation

2. Cryptography

3. Self-modification

4. Anti-analysis tricks
Anti-debugging tricks

MOV EAX,-1
INT 2E
CMP WORD PTR DS:[EDX-2],2ECD

• Call interruption 2E with an invalid EAX value

• Normal behavior: EDX contains the address of the next instruction

• Test if EDX-2 points to the opcode of INT 2E, which is 2ECD

• If a debugger is running, then EDX is equal to 0xFFFFFFFF
Consequences of Code protection

1. Difficult for a human analyst to understand a malware code

   1. Ollydbg

   2. IDA Pro
Consequences of Code protection

1. Difficult for a human analyst to understand a malware code

   1. Ollydbg
   2. IDA Pro

2. Difficult to design automatic tools

   1. Static analysis
      • Abstract interpretation
   2. Dynamic analysis
A cruel world

**Theorem**: Let \( v \) be a virus. The set of viruses which are obtained from \( v \) by mutation is not decidable (computable).

A consequence of Rice theorem

**Theorem** (RICE): Let \( P \) be a non-trivial computable property. Then the set of programs which satisfies \( P \) is not decidable.

Idea of the proof: Construct a reduction from the halting problem.
Morphological analysis in a nutshell

Signatures are abstract flow graph

Detection of subgraph in program flow graph abstraction
Automatic construction of signatures

Sample name: Email-Worm.Win32.Bagle.a
Number of nodes: 1022
Reduction of signatures by graph rewriting

<table>
<thead>
<tr>
<th>Original</th>
<th>One-one substitution</th>
<th>Substitution</th>
<th>Permutation</th>
<th>Jcc obfuscation</th>
<th>All in one</th>
<th>Normalised CFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 : cmp eax 0</td>
<td>0 : cmp eax 0</td>
<td>0 : cmp eax 0</td>
<td>0 : cmp eax 0</td>
<td>0 : cmp eax 0</td>
<td>0 : cmp eax 0</td>
<td></td>
</tr>
<tr>
<td>1 : jne +7</td>
<td>1 : jne +7</td>
<td>1 : jne +8</td>
<td>1 : jne +9</td>
<td>1 : je +2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 : mov ecx eax</td>
<td>2 : mov ecx eax</td>
<td>2 : push eax</td>
<td>2 : mov ecx eax</td>
<td>2 : jmp +10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 : dec ecx</td>
<td>3 : dec ecx</td>
<td>3 : dec ecx</td>
<td>3 : dec ecx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 : mul eax ecx</td>
<td>4 : mul eax ecx</td>
<td>4 : dec ecx</td>
<td>4 : mul eax ecx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 : cmp ecx 1</td>
<td>5 : cmp ecx 1</td>
<td>5 : mul eax ecx</td>
<td>5 : cmp ecx 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 : jne -3</td>
<td>6 : jne -3</td>
<td>6 : jae -3</td>
<td>6 : ja -3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 : jmp +2</td>
<td>7 : jmp +2</td>
<td>7 : jae -3</td>
<td>7 : ja -3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 : inc ecx</td>
<td>8 : add ecx 1</td>
<td>8 : inc ecx</td>
<td>8 : cmp ecx 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 : ret</td>
<td>10 : ret</td>
<td>10 : inc ecx</td>
<td>10 : ret</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 : ret</td>
<td>11 : ret</td>
<td>11 : inc ecx</td>
<td>11 : add ecx 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Morphological detection : Results

• False negative
  • No experiment on unknown malware
  • Signatures with < 18 nodes are potential false negative
  • Restricted signatures of 20 nodes are efficient
• Less than 3 sec. for signatures of 500 nodes
Conclusion about morphological detection

- Benchmarks are good

- Pro
  - More robust on local mutation and obfuscation
  - Detect easily variants of the same malware family
  - Try to take into account program semantics
  - Quasi-automatic generation of signatures

- Cons
  - Difficult to determine flow graph statically of self-modifying programs
  - Use of combination of static and dynamic analysis
Reference

Waledac, again ....

• How to neutralize a botnet ?

• Send the US-Marshals (see Rustock recent story)

• Design an attack

  • Good understanding of the mechanisms of a botnet (reverse engineering)

  • Find an attack

  • Large scale experiments in vitro
Waledac Peer-to-peer communication protocol

• Each peer maintains a list of known peer (RLIST)

• Bots exchange parts of their RLIST on regular basis to maintain connectivity

• Fallback mechanism over HTTP to fetch new peers

Sorted by local time

```
<lm>
<localtime>1244053204</localtime>
<nodes>
  <node ip="a.b.c.d" port="80" time="124053217">
    469abea004710c1ac0022489cef03183
  </node>
  <node ip="e.f.g.h" port="80" time="124053232">
    691775154c03424d9f12c17fdf4b640b
  </node>
...
</nodes>
</lm>
```
Waledac Peer-to-peer communication protocol

- Update are based on the most recent timestamps

- IP address does not identify a peer

- The id identifies a peer

- Vulnerability to sibyl attack

- Craft a RLIST update to put «myIP» on the top-500 list

```xml
<lm><localtime>0</localtime>
<nodes>
  <node ip="myIP" port="80" time="0">
    00000000000000000000000000000001
  </node>
  ...
  <node ip="myIP" port="80" time="0">
    00000000000000000000000000000500
  </node>
</nodes>
</lm>
```
Botnet neutralization in the lab

**Attack scenario**

- **8 PROTECTORS**
- **500 REPEATERS**
- **2500 SPAMMERS**
- **1 ATTACKER**
- **3 YBILS**
- **WHITE C&C**
A sybil attack
Spam sent by the botnet
Rlist infections for repeaters
High Security Lab @ Nancy

Telescope & honeypots
In vitro experiment clusters

lhs.loria.fr
Conclusions
Conclusion

- Mathematical definitions of malware with tools
  - High level representation of binaries
  - Abstract signature which are robust w.r.t. obfuscations
- Experiments theories
  - Analyzing tools combining static and dynamic analysis
- Detection and neutralization heuristics
Thanks !