06.16.18

Malware Analysis and Automation with Binary Ninja

Erika Noerenberg, Senior Threat Researcher

Carbon Black.

Agenda

01. What is Binary Ninja?
02. Why Binary Ninja?
03. A Brief Introduction to PlugX
04. Disassembly and IL in Binary Ninja
05. Automation of PlugX String Decryption

ABOUT ME

- **Not affiliated with Vector35**
- 10+ years in the security industry
  - Malware reverse engineering
  - Digital forensics
  - Firmware analysis
  - iOS programming
- Former Forensic Analyst and Reverse Engineer at DC3
  - Investigated defense network intrusions
    - Malware RE
    - System Forensic Analysis

Erika Noerenberg

Senior Threat Researcher

Carbon Black Threat Analysis Unit (TAU)

@gutterchurl

ACKNOWLEGEMENTS

- Everyone in binaryninja.slack.com who patiently answered questions, especially:
  - Josh Watson (@josh_watson, ToB) for the incredible API help
    - https://blog.trailofbits.com/2017/01/31/breaking-down-binary-ninjas-low-level-il/
  - Ryan Snyder (Vector35) for helping track down a core bug affecting the disassembly
  - Ryan Stortz (@withzombies, ToB) for inspiring me to try Binary Ninja
    - https://blog.trailofbits.com/2016/06/03/2000-cuts-with-binary-ninja/
  - All the early adopters in the community who have been contributing plugins and continuing discussion in slack
  - Jordan Wiens (@psifertex, Vector35) for help and support

Carbon Black.

WHAT IS BINARY NINJA?

- Reverse engineering platform developed by Vector35
- First public release (v 1.0.307) on 31 July 2016
- Robust API accessible in headless mode (not available in demo version)
- Linear disassembly, graph mode (disassembly, LLIL, MLIL)
- Fully featured hex editor with built-in transformations
- Open plugin architecture and open source community plugin repository
  - https://github.com/Vector35/community-plugins
- Active community Slack channel (#binaryninja) for announcements and support

WHY BINARY NINJA?

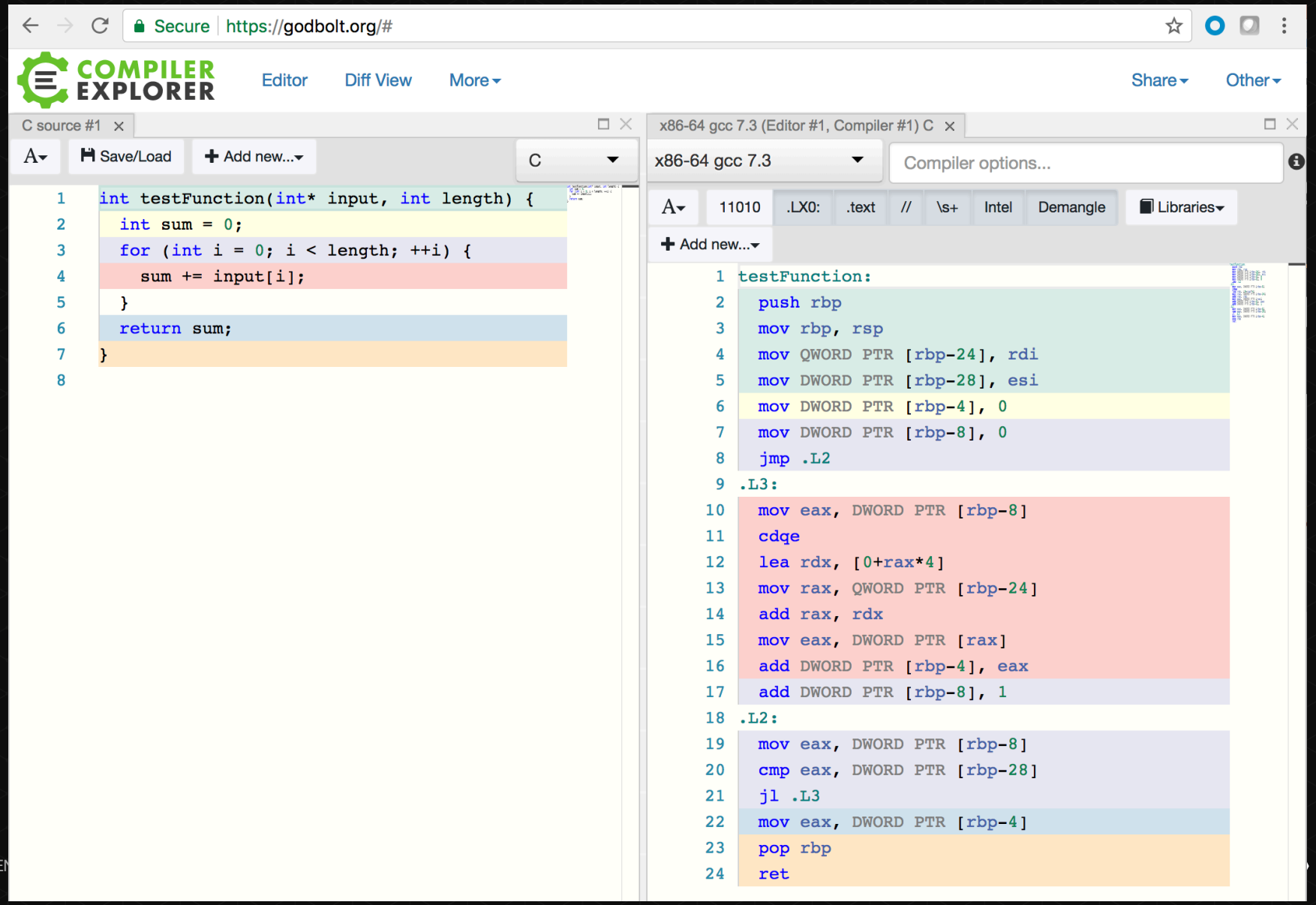
- Addition of linear sweep made malware analysis more feasible
- Free demo version and low-cost personal version make a low barrier to entering the field
  - Many trainings and university courses are now using it as a result
- Platform agility
- Support and community
- Module extensibility and community plugins
- Programming with IL (LLIL, MLIL)
- Ease of implementing new architectures

INTERMEDIATE LANGUAGES (IL) – LLIL AND MLIL

- Assembly is a low-level language that represents machine bytecode in a human-readable instruction format
- Compilers use an intermediate representation (IR) to analyze and optimize the code being compiled
  - https://blog.trailofbits.com/2017/01/31/breaking-down-binary-ninjas-low-level-il/
- Intermediate Languages provide a higher level abstraction "lifted" from assembly to read more like source code
- LLIL and MLIL are intermediate languages used by Binary Ninja (BNIL) to assist in static binary analysis
  - "The Binary Ninja Intermediate Language (BNIL) is a semantic representation of the assembly language instructions for a native architecture in Binary Ninja. BNIL is actually a family of intermediate languages that work together to provide functionality at different abstraction layers." (https://docs.binary.ninja/dev/bnil-llil/index.html)
- HLIL format in development for future release

COMPILER EXPLORER

https://godbolt.org/#



INTERMEDIATE LANGUAGES

Disassembly

```
10001008  8b350c200010      mov     esi, dword [USER32!GetDC@IAT]
1000100e  57                push    edi {var_1c}
1000100f  6a00              push    0x0 {var_20}
10001011  ffd6              call    esi
10001013  8b3d10200010      mov     edi, dword [USER32!ReleaseDC@IAT]
10001019  50                push    eax {var_24}
1000101a  6a00              push    0x0 {var_28}
1000101c  ffd7              call    edi
1000101e  8b1d00300010      mov     ebx, dword [data_10003000]
10001024  8d9b10300010      lea     ebx, [ebx+0x10003010]
1000102a  6a00              push    0x0 {var_2c}
1000102c  895df8            mov     dword [ebp-0x8 {var_c}], ebx
1000102f  ffd6              call    esi
10001031  50                push    eax {var_30}
10001032  6a00              push    0x0 {var_34}
10001034  ffd7              call    edi
10001036  8b4d0c            mov     ecx, dword [ebp+0xc {arg2}]
10001039  85c9              test    ecx, ecx
1000103b  7e3e              jle     0x1000107b
```

LLIL Representation

```
 5 @ 10001008  esi = [USER32!GetDC@IAT].d
 6 @ 1000100e  push(edi)
 7 @ 1000100f  push(0)
 8 @ 10001011  call(esi)
 9 @ 10001013  edi = [USER32!ReleaseDC@IAT].d
10 @ 10001019  push(eax)
11 @ 1000101a  push(0)
12 @ 1000101c  call(edi)
13 @ 1000101e  ebx = [data_10003000].d
14 @ 10001024  ebx = ebx + 0x10003010
15 @ 1000102a  push(0)
16 @ 1000102c  [ebp - 8 {var_c}].d = ebx
17 @ 1000102f  call(esi)
18 @ 10001031  push(eax)
19 @ 10001032  push(0)
20 @ 10001034  call(edi)
21 @ 10001036  ecx = [ebp + 0xc {arg2}].d
22 @ 1000103b  if (ecx s<= 0) then 23 @ 0x1000107b else 48 @ 0x1000103d
```

MLIL Representation

```
 5 @ 10001011  eax = USER32!GetDC(0)
 6 @ 10001019  int32_t var_24 = eax
 7 @ 1000101a  int32_t var_28 = 0
 8 @ 1000101c  USER32!ReleaseDC(0, var_24)
 9 @ 1000101e  int32_t ebx_1 = [data_10003000].d
10 @ 10001024  int32_t ebx_2 = ebx_1 + 0x10003010
11 @ 1000102a  int32_t var_2c = 0
12 @ 1000102c  int32_t var_c = ebx_2
13 @ 1000102f  eax_1 = USER32!GetDC(0)
14 @ 10001031  int32_t var_30 = eax_1
15 @ 10001032  int32_t var_34 = 0
16 @ 10001032  int32_t* esp_1 = &var_34
17 @ 10001034  USER32!ReleaseDC(0, var_30)
18 @ 10001036  int32_t ecx = arg2
19 @ 1000103b  if (ecx s<= 0) then 20 @ 0x1000107b else 55 @ 0x1000103d
```

```
mov    esi, [GetDC]
push   0x0
call   esi
mov    edi, [ReleaseDC]
push   eax
push   0x0
call   edi
```

```
esi = [GetDC]
push(0)
call(esi)
edi = [ReleaseDC]
push(eax)
push(0)
call(edi)
```

```
eax = GetDC(0)
var1 = eax
var2 = 0
ReleaseDC(0, var1)
```


Carbon Black.

Original Disassembly

```
mov     esi, dword [USER32!GetDC@IAT]
push    edi {var_1c}
push    0x0 {var_20}
call    esi
mov     edi, dword [USER32!ReleaseDC@IAT]
push    eax {var_24}
push    0x0 {var_28}
call    edi
mov     ebx, dword [data_10003000]
lea     ebx, [ebx+0x10003010]
push    0x0 {var_2c}
mov     dword [ebp-0x8 {var_c}], ebx
call    esi
push    eax {var_30}
push    0x0 {var_34}
call    edi
mov     ecx, dword [ebp+0xc {arg2}]
test    ecx, ecx
jle     0x1000107b
```

LLIL Representation

```
esi = [USER32!GetDC@IAT].d
push(edi)
push(0)
call(esi)
edi = [USER32!ReleaseDC@IAT].d
push(eax)
push(0)
call(edi)
ebx = [data_10003000].d
ebx = ebx + 0x10003010
push(0)
[ebp - 8 {var_c}].d = ebx
call(esi)
push(eax)
push(0)
call(edi)
ecx = [ebp + 0xc {arg2}].d
if (ecx s<= 0) then 23 @ 0x1000107b else 48 @ 0x1000103d
```


Carbon Black.

LLIL Representation

```
esi = [USER32!GetDC@IAT].d
push(edi)
push(0)
call(esi)
edi = [USER32!ReleaseDC@IAT].d
push(eax)
push(0)
call(edi)
ebx = [data_10003000].d
ebx = ebx + 0x10003010
push(0)
[ebp - 8 {var_c}].d = ebx
call(esi)
push(eax)
push(0)
call(edi)
ecx = [ebp + 0xc {arg2}].d
if (ecx s<= 0) then 23 @ 0x1000107b else 48 @ 0x1000103d
```

MLIL Representation

```
eax = USER32!GetDC(0)
int32_t var_24 = eax
int32_t var_28 = 0
USER32!ReleaseDC(0, var_24)
int32_t ebx_1 = [data_10003000].d
int32_t ebx_2 = ebx_1 + 0x10003010
int32_t var_2c = 0
int32_t var_c = ebx_2
eax_1 = USER32!GetDC(0)
int32_t var_30 = eax_1
int32_t var_34 = 0
int32_t* esp_1 = &var_34
USER32!ReleaseDC(0, var_30)
int32_t ecx = arg2
if (ecx s<= 0) then 20 @ 0x1000107b else 55 @ 0x1000103d
```


Carbon Black.

Original Disassembly

```
mov     esi, dword [USER32!GetDC@IAT]
push    edi {var_1c}
push    0x0 {var_20}
call    esi
mov     edi, dword [USER32!ReleaseDC@IAT]
push    eax {var_24}
push    0x0 {var_28}
call    edi
mov     ebx, dword [data_10003000]
lea     ebx, [ebx+0x10003010]
push    0x0 {var_2c}
mov     dword [ebp-0x8 {var_c}], ebx
call    esi
push    eax {var_30}
push    0x0 {var_34}
call    edi
mov     ecx, dword [ebp+0xc {arg2}]
test    ecx, ecx
jle     0x1000107b
```

MLIL Representation

```
eax = USER32!GetDC(0)
int32_t var_24 = eax
int32_t var_28 = 0
USER32!ReleaseDC(0, var_24)
int32_t ebx_1 = [data_10003000].d
int32_t ebx_2 = ebx_1 + 0x10003010
int32_t var_2c = 0
int32_t var_c = ebx_2
eax_1 = USER32!GetDC(0)
int32_t var_30 = eax_1
int32_t var_34 = 0
int32_t* esp_1 = &var_34
USER32!ReleaseDC(0, var_30)
int32_t ecx = arg2
if (ecx s<= 0) then 20 @ 0x1000107b else 55 @ 0x1000103d
```


Carbon Black.

BINARY VIEW

- Raw vs. PE

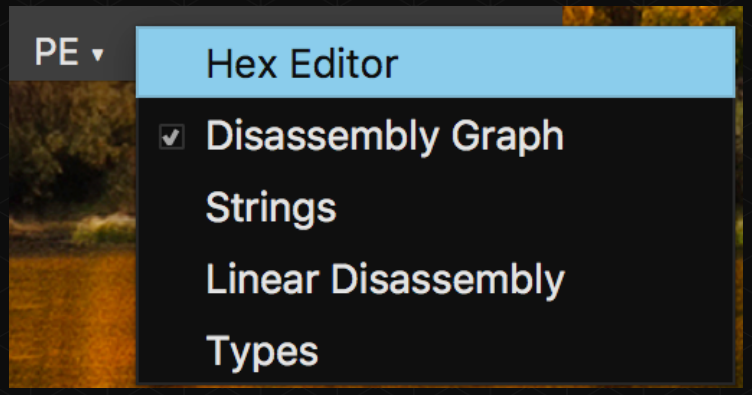
binaryview module

`binaryninja.binaryview.AddressRange` (start, end)	
`binaryninja.binaryview.AnalysisCompletionEvent` (...)	The `AnalysisCompletionEvent` object provides an asynchronous mechanism for receiving callbacks when analysis is complete.
`binaryninja.binaryview.AnalysisProgress` (...)	
`binaryninja.binaryview.BinaryDataNotification` ()	
`binaryninja.binaryview.BinaryDataNotificationCallbacks` (...)	
`binaryninja.binaryview.BinaryReader` (view[, ...])	`class BinaryReader` is a convenience class for reading binary data.
`binaryninja.binaryview.BinaryView` ([...])	`class BinaryView` implements a view on binary data, and presents a queryable interface of a binary file. One key
`binaryninja.binaryview.BinaryViewType` (handle)	
`binaryninja.binaryview.BinaryWriter` (view[, ...])	`class BinaryWriter` is a convenience class for writing binary data.
`binaryninja.binaryview.DataVariable` (addr, ...)	
`binaryninja.binaryview.Section` (name, ...)	
`binaryninja.binaryview.Segment` (start, ...)	
`binaryninja.binaryview.StringReference` (bv, ...)	

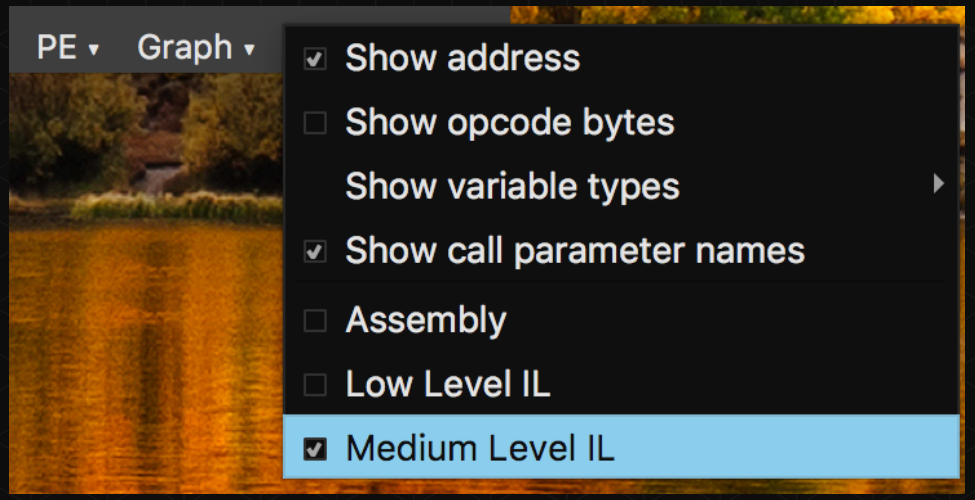
Carbon Black.

LET'S TAKE A CLOSER LOOK…

- Menus – bottom right corner (UI overhaul coming in v1.2)
- Types view
- Hex editor



- LLIL/MLIL



Carbon Black.

WHAT IS PLUGX?

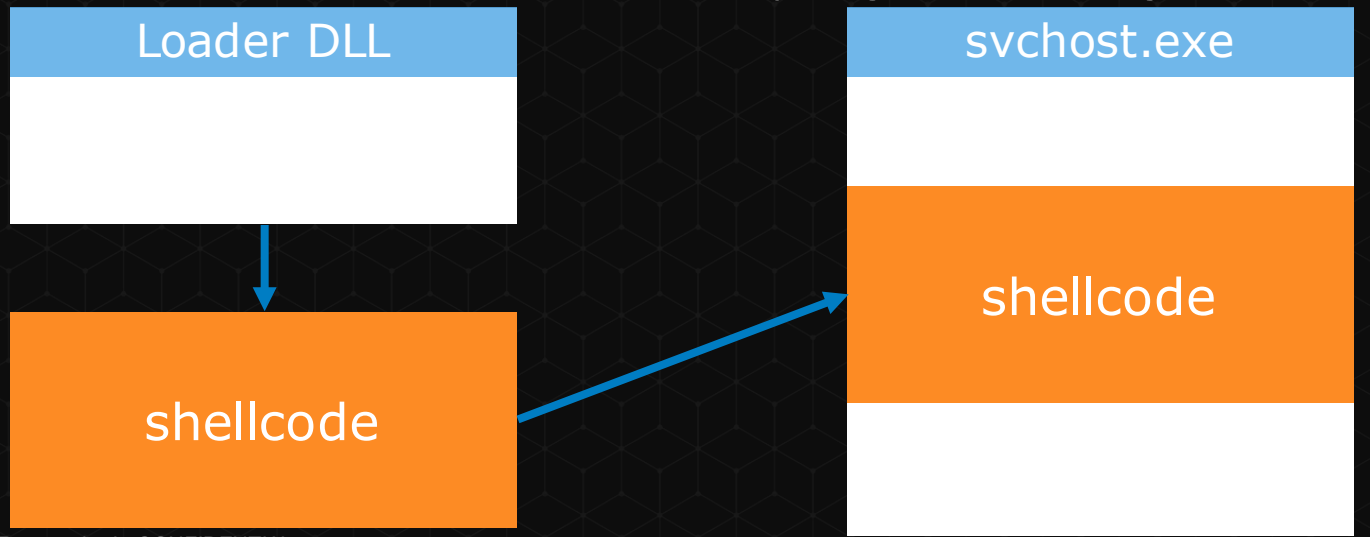
- Fully-featured RAT (Remote Access Trojan/Remote Administration Tool)
- First samples seen ~2008, still used in current campaigns today
- Although there has been code evolution among “variants,” most have followed a similar infection methodology through the years
  - Either self-extracting RAR (SFX) or dropper creates three files:
    - Legitimate, signed executable
    - DLL Loader, “sideloaded” by legitimate executable
    - Encrypted shellcode
  - Shellcode typically injected into legitimate system process
- The sample I will use for today’s demonstration follows this pattern

PLUGX: SIDELOADING

- Legitimate Windows executables almost always rely on external libraries (DLLs) for certain functionality
  - These libraries are loaded at runtime
  - The executable may not fully validate the DLL before loading it
- Malware can take advantage of this by tricking the legitimate app into loading the malicious DLL rather than the expected one
- Often, malware authors will use legitimate executables from antivirus or other security products to avoid suspicion or take advantage of whitelisting
  - AShld.exe / AShldRes.dll – McAfee VShield component
  - fsguidll.exe / fslapi.dll – FSecure GUI component
  - OINFO11.exe / Oinfo11.ocx – Office 2003 SP2 Update

PLUGX: PROCESS HOLLOWING

- Third component of the standard PlugX variant contains encrypted, compressed shellcode and C2/configuration data
- The loader DLL decrypts and decompresses the shellcode
  - Loader starts a new instance of a process (such as svchost.exe) in a suspended state
  - Executable memory region is freed or unmapped by loader
  - Loader maps shellcode into freed memory region and begins execution



Carbon Black.

PLUGX: DECRYPTION METHOD

- Find decryption routine via YARA hit
  - YARA: signatures based on pattern matching

```
16      strings:
17          $byte1 = { 8A [2-4] 8A [2-4] FF 05 00 30 00 10 [0-5] 2A [1-6] 80 [2-7] 02 [1-6] 88 0? }
18          $byte2 = { 8B [2-4] 8A [2-4] FF 05 00 30 00 10 [0-5] 2A [1-6] 80 [2-7] 02 [1-6] 88 0? }
```

- Get the offset of the decryption code
- Get cross references (xrefs) to the decryption function
- Get parameters passed into decryption function
- Decrypt strings and patch binary

Carbon Black.

MAIN ROUTINE

```
def decode_strings(filename):
    y_offsets = get_yara_offset(filename)
    bv = BinaryViewType.get_view_of_file(filename)
    bv.add_analysis_option("linearsweep")
    bv.update_analysis_and_wait()
    xref_funcs = get_xrefs(bv, y_offsets)
    keys = get_key(bv, y_offsets)
    # Need to make this work for multiple keys but not y_offset
    XOR_KEY = keys[0].value.value
    #for y in y_offsets:
    #    xor_dec(XOR_KEY, xref_funcs, y_off
    xor_dec(bv, XOR_KEY, xref_funcs, y_offsets)
    bv.save('/malwarz/PlugX/a96dd/AShldRes_mod.dll')
    print bv.get_strings(0x10002030, 200)
```

GET YARA HIT AND OFFSET

```
def get_yara_offset(filename):
    y_offsets = []
    hit_offsets = yara_hit(filename)
    for hits in hit_offsets:
        y_offset = hits[0]
        y_offsets.append(y_offset)
    if not y_offsets:
        log.log_error("No yara hit offsets")
        sys.exit(1)
    else:
        return y_offsets

def yara_hit(filename):
    hit_offsets = []
    matches = rules.match(filename)
    for match in matches:
        hit_offset = match.strings[0]
        hit_offsets.append(hit_offset)
    if not hit_offsets:
        log.log_error("No yara hits")
        sys.exit(1)
    else:
        return hit_offsets
```


Carbon Black.

GET XREFS TO DECRYPTION FUNCTION

```
def get_xrefs(bv, y_offsets):
    xref_funcs = []
    hit_funcs = []
    for y in y_offsets:
        hit_init = bv.get_address_for_data_offset(y)
        hit_funcs.append(bv.get_functions_containing(hit_init))

    for func_list in hit_funcs:
        for func in func_list:
            xref_funcs.append(bv.get_code_refs(func.start))
    return xref_funcs
```


Carbon Black.

GET XOR KEY FOR DECRYPTION

```python
def get_key(bv, y_offsets):
    keys = []
    fn = None
    il_bb = None
    for y in y_offsets:
        virt_off = bv.get_address_for_data_offset(y)
        fn = bv.get_functions_containing(virt_off)[0]
        fn_blocks = bv.get_basic_blocks_at(virt_off)
        il = fn.get_low_level_il_at(fn_blocks[0].start)
        for bb in fn.low_level_il:
            if bb.start == il.instr_index:
                il_bb = bb
                break
        for il in il_bb:
            if (il.operation == LowLevelILOperation.LLIL_SET_REG and
                il.src.operation == LowLevelILOperation.LLIL_XOR):
                keys.append(il.src.right)
                break
        return keys
```

Disassembly

```
10001060  sub     cl, al
10001062  xor     cl, 0x3f
10001065  add     cl, al
```

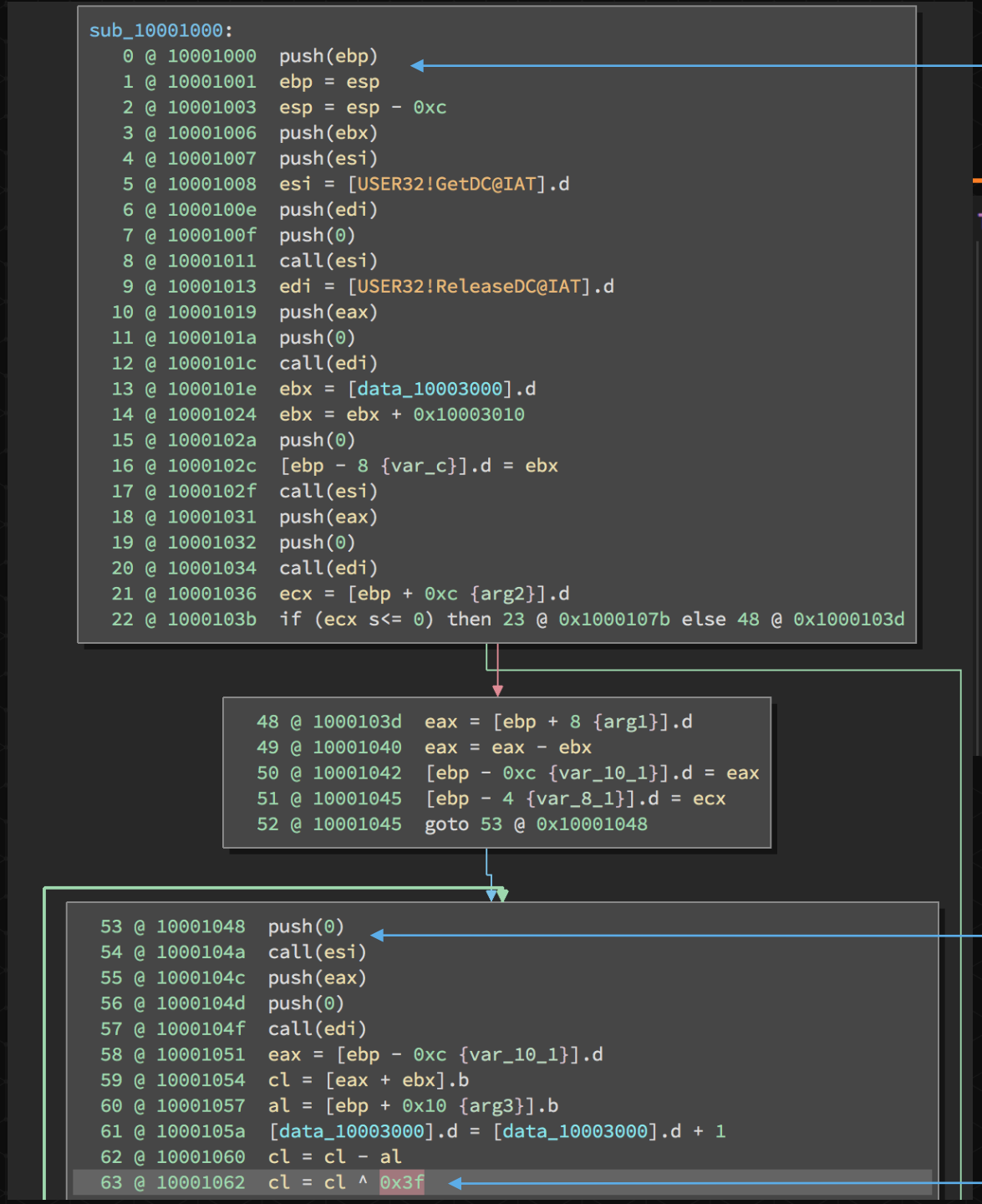
LLIL

```
62 @ 10001060  cl = cl - al
63 @ 10001062  cl = cl ^ 0x3f
64 @ 10001065  cl = cl + al
```

MLIL

```
ecx_1.cl = ecx_1.cl - eax_5.al
ecx_1.cl = ecx_1.cl ^ 0x3f
ecx_1.cl = ecx_1.cl + eax_5.al
```


Carbon Black.



Start of function

```
for y in y_offsets:
    virt_off = bv.get_address_for_data_offset(y)
    fn = bv.get_functions_containing(virt_off)[0]
    fn_blocks = bv.get_basic_blocks_at(virt_off)
    il = fn.get_low_level_il_at(fn_blocks[0].start)
    for bb in fn.low_level_il:
        if bb.start == il.instr_index:
            il_bb = bb
            break
    for il in il_bb:
        if (il.operation == LowLevelILOperation.LLIL_SET_REG and
            il.src.operation == LowLevelILOperation.LLIL_XOR):
            keys.append(il.src.right)
```

Start of “basic block”

Location of target code

Carbon Black.

DECRYPT STRINGS AND PATCH BINARY

```
def xor_dec(bv, key, xref_funcs, y_offsets):
    br = BinaryReader(bv)
    bw = BinaryWriter(bv)
    print "Decrypted strings:"
    for xref in xref_funcs[0]:
        # print xref.function, hex(xref.address)
        il = xref.function.get_low_level_il_at(xref.address).medium_level_il
        if (il.operation == MediumLevelILOperation.MLIL_CALL):
            enc_str = il.params[0].value.value
            str_size = il.params[1].value.value
            diff = il.params[2].value.value
            dec_str = ''
            br.seek(enc_str)
            bw.seek(enc_str)
            for i in xrange(str_size):
                enc_byte = br.read8()
                x = ((enc_byte - diff) ^ key) + diff
                dec_str = dec_str + chr(x)
                # Patch binary
                bw.write8(x)
```

Carbon Black.

DECRYPT STRINGS AND PATCH BINARY

```
def xor_dec(bv, key, xref_funcs, y_offsets):
    br = BinaryReader(bv)
    bw = BinaryWriter(bv)
    print "Decrypted strings:"
    for xref in xref_funcs[0]:
        # print xref.function, hex(xref.address)
        il = xref.function.get_low_level_il_at(xref.address).medium_level_il
        if (il.operation == MediumLevelILOperation.MLIL_CALL):
            enc_str = il.params[0].value.value
            str_size = il.params[1].value.value
            diff = il.params[2].value.value
            dec_str = ''
            br.seek(enc_str)
            bw.seek(enc_str)
            for i in xrange(str_size):
                enc_byte = br.read8()
                x = ((enc_byte - diff) ^ key) + diff
                dec_str = dec_str + chr(x)
                # Patch binary
                bw.write8(x)
```

```
int32_t var_60 = 0x95
int32_t var_64 = 0x12
int32_t var_68 = 0x10002028
eax_7 = dec_strings(0x10002028, 0x12, 0x95)
```

params[]

Carbon Black.

DECRYPT STRINGS AND PATCH BINARY

```
def xor_dec(bv, key, xref_funcs, y_offsets):
    br = BinaryReader(bv)
    bw = BinaryWriter(bv)
    print "Decrypted strings:"
    for xref in xref_funcs[0]:
        # print xref.function, hex(xref.address)
        il = xref.function.get_low_level_il_at(xref.address).medium_level_il
        if (il.operation == MediumLevelILOperation.MLIL_CALL):
            enc_str = il.params[0].value.value
            str_size = il.params[1].value.value
            diff = il.params[2].value.value
            dec_str = ''
            br.seek(enc_str)
            bw.seek(enc_str)
            for i in xrange(str_size):
                enc_byte = br.read8()
                x = ((enc_byte - diff) ^ key) + diff
                dec_str = dec_str + chr(x)
                # Patch binary
                bw.write8(x)
```

```
int32_t var_60 = 0x95
int32_t var_64 = 0x12
int32_t var_68 = 0x10002028
eax_7 = dec_strings(0x10002028, 0x12, 0x95)
```

params[]

Carbon Black.

FUTURE WORK

- Mac/iOS analysis, automation, and sandboxing
- Emulator (extend emilator? https://github.com/joshwatson/emilator)
  - Automated malware unpacking/deobfuscation
- Fileinfo / Noriben (https://github.com/Rurik/Noriben)
- Using IL to identify code patterns as with YARA
  - Abstraction of assembly to better identify custom routines?

Carbon Black.

Q&A?

- Contact info:
  - Twitter: @gutterchurl
  - Email: enoerenberg@carbonblack.com

Carbon Black.

RESOURCES

Binary Ninja Demo version (free!)
https://binary.ninja/demo/

Binary Ninja GitHub
https://github.com/Vector35

Binary Ninja slack channel
https://binaryninja.slack.com

Binary Ninja API
https://api.binary.ninja

Community plugins
https://github.com/Vector35/community-plugins/tree/master/plugins

PlugX overview and resources
https://logrhythm.com/blog/deep-dive-into-plugx-malware/

Carbon Black.

www.CarbonBlack.com

Thank you.

enoerenberg@carbonblack.com
@gutterchurl
LinkedIn: Erika Noerenberg

Carbon Black.

Carbon Black.

www.CarbonBlack.com