Behavior-Driven Development in Malware Analysis

Thomas Barabosch, Elmar Gerhards-Padilla
firstname.lastname@fkie.fraunhofer.de
Botconf 2015, Paris, France

Cyber Analysis & Defense (CA&D)
Motivation

- Malware analysis continues to be a tedious and time consuming task (some might call it job security…)
- Extraction of malicious behavior is a daily task
  - Analyze (obfuscated) binary code
  - Reimplement in higher language like Python or C (Reimplementation task)
- Code is just “translated” from assembly to higher language
  - Functionality is not ensured
  - Readability is poor
  - No documentation
  - Underlying semantics not clear

Solution: Improve current process
Related Work

- Extraction of malicious behavior
  - [Caballero2010], [Kolbitsch2010], [Barabosch2012]
- Using TDD in RE processes
  - [VanLindberg2008], [DeSousa2010]

*However*, current state-of-the-art solutions
- are not publicly available
- can not cope with anti-analysis techniques
- can not cope with complex obfuscations
- assume source code and documentation available
Requirements of Solution

1. Allows the analyst to describe concisely and naturally what he observes
2. Ensures that the code works continuously during the implementation
3. Resulting code should be concise, documented and readable
4. Increases the focus of the analyst

Proposed Solution:
Apply Behavior-Driven Development to Malware Analysis
*-DRIVEN DEVELOPMENT
In the Beginning there was Software Testing...

- Tests whether a software does what it is supposed to do
- Shows quality of a software to stake-holders
- Finds defects and failures in a software

Problems

- Infrequent testing (e.g. Waterfall model)
- Code coverage
- Not efficient if done manually
Test Driven Development (TDD)

- Short development cycle
- Ideally ensures 100% coverage
- Small and comprehensive code base due to frequent refactoring
- Tests serve as a documentation of the code

The mantra of Test-Driven Development (TDD) is “red, green, refactor.”
Source: http://luizricardo.org/wordpress/wp-content/upload-files/2014/05/tdd_flow.gif
Behavior Driven Development (BDD)

- BDD focuses on a clear understanding of the software’s behavior rather than modules, functions, etc.
- BDD emerged from TDD
- Test cases are formulated in natural language
- Strong theoretical foundation (Hoare logic)
  - \( \{P\} C \{Q\} \rightarrow \text{Given } _\_ \text{ When } _\_ \text{ Then } _\_ \)
Behavior Driven Development (BDD)

**Scenario:** Coffee maker can add sugar to coffee

*Given* customer chooses sugar

*When* customer presses OK button

*Then* coffee maker adds sugar to coffee
BDD IN MALWARE ANALYSIS
Overview of the Process

- Preparation phase
- Implementation phase (Observe – Test – Code – Refactor)
Preparation - Pinpointing the Behavior

- First pinpoint the behavior in the binary
  - Find entry point \( S \) and exits \( \{E_1, \ldots, E_n\} \)
  - Extract initial test data for acceptance test
  - State acceptance test

Pinpointing the Behavior (DGA)

- **Domain Generation Algorithm**
  - See Daniel’s talk *(DGArchive – A deep dive into domain generating malware)*

- Several types of DGAs [Barabosch2012]
  - Deterministic/non-deterministic
  - Time-dependent/independent

- Naïve approach (forwards): look for timing sources
  - E.g. *GetSystemTime, NtQuerySystemTime, GetLocalTime*

- Naïve approach (backwards): DNS resolution
  - E.g. *gethostbyname*
Pinpointing the Behavior (command dispatcher)

- Bots implement several commands
- Bots receive and process messages of botmaster
- Command dispatcher
- Naïve approach: follow data flow from network source
- Monitor networking APIs like receive
- Follow data flow in forwards direction until switch statement

Example: Dridex
Preparation - Initial End-To-End Acceptance Test

- Serves as guide throughout the implementation phase
- Tests behavior as a black box
- Capture data at $S$ and $\{E_1, \ldots, E_n\}$
- Once this test passes -> reimplementation successfully
Step 1: Observing the Behavior

- Top-Down-Approach
  - Getting a rough overview
  - Identifying individual features and their interfaces

- Gather test data at interfaces (input/output)
  - Use this data for mocking later
  - Mock interfaces of submodules at first
Step 2: Writing a Test

- **Given-Then-When**

- **Fundamental: mock objects**
  - Mimic the behavior of real objects
  - In software development, they replace, e.g., non-existing objects
  - In our case, they replace modules that are not 100% understood
  - Gather test data at module interfaces
Step 2: Writing a Test

- **Given-Then-When**

- **Fundamental: mock objects**
  - Mimic the behavior of real objects
  - In software development, they replace, e.g., non-existing objects
  - In our case, they replace modules that are not 100% understood
  - Gather test data at module interfaces
Step 2: Writing a Test

- Given-Then-When

- Fundamental: mock objects
  - Mimic the behavior of real objects
  - In software development, they replace, e.g., non-existing objects
  - In our case, they replace modules that are not 100% understood
  - Gather test data at module interfaces
Step 3: Making the Test Pass

- Just write enough code to make the test pass
- Binary serves as valid system specification
- Focus and just implement code to make the test pass
- "Premature optimization is the root of all evil"
Step 4: Refactoring the Code

- Altering the syntax without altering the semantics
- Ensures conciseness and readability
- Many refactorings do exist (see also [Fowler1999])
  - Refactoring inlined code (memcpy)
  - Break up complex expressions
  - Removing dead expressions
- Does the end-to-end acceptance test pass?
Limitations

- Decrease in time efficiency
  - Extra time pays off due to benefits
  - TDD comes with an overhead of 15% to 35% [Bhat2006]
- TDD/BDD comes from “normal” software development
  - Reusability not needed in malware analysis
  - Long-running projects do exist also in the field of malware analysis
CASE STUDY NYMAIM DGA
Nymaim

- Nymaim is a malware dropper
  - But also credential stealer, SOCKS, etc.
- Heavily obfuscated
  - Decompilers fail to work
  - See IDApatchwork presentation of Daniel Plohmann
Unpacked Dridex

Regular functions

No strange constants

Resolved imports

Reasonable control flow

...
; Attributes: bp-based frame
sub_4617B72 proc near

arg_0= dword ptr 8
arg_4= dword ptr 0Ch

; FUNCTION CHUNK AT seg000:00003406 SI:
; FUNCTION CHUNK AT seg000:0000FF1 SI:
; FUNCTION CHUNK AT seg000:00014729 SI:

push    ebp
mov     ebp, esp
push    eax
push    ecx
jmp     loc_46034D6
sub_4617B72 endp

; ================================
lea     esi, [ebp-1Ch]
push    63h ; 'c'
call    sub_460A4C2
push    ecx
push    66E7E05Bh
push    66E082D2Ch
call    sub_460C4AC8
mov     ecx, [esi]
add     ecx, [esi+4]
mov     eax, 99ADDFF81h
call    sub_461AB04
add     eax, ecx
mov     [ebp-2Ch], eax
mov     eax, 9FA6BD27h
call    sub_461AB04
add     eax, ecx
mov     [ebp-28h], eax
mov     eax, 9F3EAD68h
push    edx
call    sub_4603580
cvtps2pd xmm2, xmm3
pop     ecx

; =============== SUBROUTINE
Unpacked Nymaim

Irregular functions

Function entries

Function ends

Strange constants
Unpacked Nymaim

Irregular functions

Function entries

Function ends

Strange constants

Control flow computed dynamically
Unpacked Nymaim

Irregular functions

Function entries

Function ends

Strange constants

Control flow computed dynamically

Confuses disassembler

; Attributes: bp-based frame

sub_4617B72 proc near

arg_0= dword ptr 8
arg_4= dword ptr 0Ch

; FUNCTION CHUNK AT seg000:000034D6 SI:
; FUNCTION CHUNK AT seg000:00000FF1 SI:
; FUNCTION CHUNK AT seg000:00014729 SI:

push ebp
mov ebp, esp
push eax
push ecx
jmp loc_46034D6
sub_4617B72 endp

; -------------------------------------
leal esi, [ebp-1Ch]
push 63h ; 'c'
call sub_460A4C2
push ebx
push 66E7E05Bh
push 66E82D2Ch
call sub_460C4CB
mov ecx, [esi]
add ecx, [esi+4]
mov eax, 99ADDFFB1h
call sub_461AB04
add eax, ecx
mov [ebp-2Ch], eax
mov eax, 9FA68D27h
call sub_461AB04
add eax, ecx
mov [ebp-28h], eax
mov eax, 9F3EAD68h
push 8A62BDC07h
call sub_462F5F8
cvpstpd xmm2, xmm3
mov ecx

; =========== SUBROUTINE ===========
Nymaim’s DGA – Tools of Trade and Resources

- **Tools of trade**
  - Immunity Debugger 1.85
  - IDA Pro 6.8
  - Mandiant ApateDNS 1.0
  - Python 2.7.9
  - Behave 1.2.5 [Behave2015]

- **Source code on Bitbucket!**
Nymaim’s DGA – First Observations

- Black-boxing shows that
  - At first four hard-coded domain are resolved and contacted

<table>
<thead>
<tr>
<th>Time</th>
<th>Domain Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>06:17:03</td>
<td>google.com</td>
</tr>
<tr>
<td>06:17:03</td>
<td>timetengstell.com</td>
</tr>
<tr>
<td>06:17:04</td>
<td>timetengstell.com</td>
</tr>
<tr>
<td>06:17:05</td>
<td>timetengstell.com</td>
</tr>
<tr>
<td>06:17:06</td>
<td>timetengstell.com</td>
</tr>
<tr>
<td>06:17:07</td>
<td>fnpoxe.xyz</td>
</tr>
<tr>
<td>06:17:08</td>
<td>fexfmywazxk.net</td>
</tr>
<tr>
<td>06:17:09</td>
<td>pdudefhfb.net</td>
</tr>
<tr>
<td>06:17:10</td>
<td>dvkbdi.xyz</td>
</tr>
<tr>
<td>06:17:11</td>
<td>vsikbrtmbsm.xyz</td>
</tr>
<tr>
<td>06:17:12</td>
<td>ntpervk.info</td>
</tr>
</tbody>
</table>
Nymaim’s DGA – First Observations

- Black-boxing shows that
  - At first four hard-coded domain are resolved and contacted
  - In case of failure domains are generated and resolved
- **Deterministic**: same results in two different VMs
- **Time-dependent**: different results when date changed
Nymaim’s DGA – First Observations

- Black-boxing shows that
  - At first four hard-coded domain are resolved and contacted
  - In case of failure domains are generated and resolved
- **Deterministic**: same results in two different VMs
- **Time-dependent**: different results when date changed

Pinpointing the algorithm
- Breaking on `GetSystemTime` -> Bingo!
- Input: time
- Output: 30 domain names
Nymaim’s DGA – Our First Test: Acceptance Test

- We know already many important parameters
  - Interfaces of algorithm
- Also we have gathered a first set of test data
  - Time information and list of generated domains
- We write our first end-to-end acceptance test
  - It does not pass
  - However, once it passes we are done!
Nymaim’s DGA – Our First Test: Acceptance Test

Scenario: Nymaim DGA computes domains of 2015-06-12
Given the day is "2015-06-12"
When DGA computes domains for this date
Then the domains for this date are

<table>
<thead>
<tr>
<th>domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmjdfotcy.in</td>
</tr>
<tr>
<td>yjcmub.info</td>
</tr>
<tr>
<td>uiismpexr.info</td>
</tr>
<tr>
<td>rszsgpzivi.info</td>
</tr>
</tbody>
</table>
Nymaim’s DGA – Our First Test: Acceptance Test

Scenario: Nymaim DGA computes domains of 2015-06-12
Given the day is "2015-06-12"
When DGA computes domains for this date
Then the domains for this date are

<table>
<thead>
<tr>
<th>domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmjdfotcy.in</td>
</tr>
<tr>
<td>yjcmub.info</td>
</tr>
<tr>
<td>uiismpexr.info</td>
</tr>
<tr>
<td>rszsgpzivi.info</td>
</tr>
</tbody>
</table>

Failing scenarios:
features/dga.feature:5 Nymaim DGA computes domains of 2015-06-12

0 features passed, 1 failed, 0 skipped
0 scenarios passed, 1 failed, 0 skipped
2 steps passed, 1 failed, 0 skipped, 0 undefined
Took 0m0.002s
Nymaim’s DGA – Our First Test: Acceptance Test

Scenario: Nymaim DGA computes domains of 2015-06-12
Given the day is "2015-06-12"
When DGA computes domains for this date
Then the domains for this date are

domains
| dmjdfotcy.in |
yjcmub.info |
| uiismpexr.info |
| rszsgpzivi.info |

Failing scenario
features/d
0 features passed
0 scenarios
2 steps passed
Took 0m0.002s

FAIL
Nymaim’s DGA – Overview

- While stepping over the code we have noticed that there
  - Initialization
  - Main logic
  - PRNG (Xorshift)

- We focus on one component at a time
  - Reverse the main logic, mock the rest!
Nymaim’s DGA – Main Logic

```assembly
push    duord ptr [ebp-30h]
push    6
push    edx
push    0160F558h
push    6f959f1f2h
call   sub_4612835
lea    ecx, [eax+6]
lea    ebx, [esi+4]

loc_46162A8:
call   sub_461281D
push    duord ptr [ebp-30h]
push    50h ; 'j'
call   obfuscateRegisterPush
push    edx
push    9999512h
push    6765628h
call   sub_46129EE
mov    [ebx], al
call   sub_4613802
add    [ebx], al
inc    ebx
dec    ecx
jnz    short loc_46162A8
call   sub_4612912
mov    [ebx], al
inc    ebx
push    duord ptr [ebp-30h]
push    6
push    esi
push    56019402h
push    5A020D2h
Call   sub_4614592
inc    eax
dec    eax
jz     tld_ru
dec    eax
jz     tld_net
dec    eax
jz     tld_in
dec    eax
jz     tld_con
dec    eax
jz     tld_xyr
call   obfuscateString
mov    [ebx], eax
mov    duord ptr [ebx+4], 0
add    ebx, 5

loc_4616326:
lea    eax, [esi+4]
sub    ebx, esi
mov    [esi+2], bx
add    esi, ebx
dec    duord ptr [ebp-8]
inc    loc_4614200
```
Nymaim's DGA – Main Logic

```
push    duword ptr [ebp-30h]
push    6
push    edx
push    0160F530h
push    6E9591F2h
call    sub_A601335
lea     ecx, [eax+6]
lea     ebx, [esi+4]

loc_46162A8:
call    sub_A60591ED
push    duword ptr [ebp-30h]
push    50h ; '
call    obfuscrateRegisterPush
push    edx
push    980951E2h
push    67863528h
call    sub_A6829EF
mov     [ebx], al
call    sub_A603802
add     [ebx], al
inc     ebx
dec     ecx
jnz     short loc_46162A8
call    sub_A601912
mov     [ebx], al
inc     ebx
push    duword ptr [ebp-30h]
push    6
push    esi
push    56D19CB2h
push    56D200F2h
call    sub_A6014592
inc     eax
dec     eax
jz      tld_ru
dec     eax
jz      tld_net
dec     eax
jz      tld_in
dec     eax
jz      tld_com
dec     eax
jz      tld_xyr
call    obfuscrateString
mov     [ebx], eax
mov     duword ptr [ebx+4], 0
add     ebx, 5

loc_4616326:
lea     eax, [esi+4]
sub     ebx, esi
mov     [esi+2], bx
add     esi, ebx
dec     duword ptr [ebp-8]
inx     loc_A414260
```
Nymaim’s DGA – Main Logic

```assembly
push    DWORD PTR [EBP-30H]
push    6
push    EDX
push    016F538bh
push    66958F22h
call    SUB_4607305
lea     ECX, [EAX+6]
lea     EBX, [ESI+4]
loc_46162A8:
call    SUB_46071ED
push    DWORD PTR [EBP-30H]
push    50h ; '}
call    srf64RegisterPush
push    EDX
push    890551E2h
push    67B63528h
call    SUB_46829EF
mov     [EBX], AL
call    SUB_4613B02
add     [EBX], AL
inc     EBX
dec     ECX
jnz     short loc_46162A8
call    SUB_4607012
mov     [EBX], AL
inc     EBX
push    DWORD PTR [EBP-30H]
push    6
push    ESI
push    56D9F002h
push    56D200F2h
call    SUB_4614592
inc     EAX
dec     EAX
jz      tld_ru
dec     EAX
jz      tld_net
dec     EAX
jz      tld_in
dec     EAX
jz      tld_com
dec     EAX
jz      tld_net
call    deobfuscateString
mov     [EBX], EAX
mov     DWORD PTR [EBX+4], 0
add     EBX, 5
loc_461692E:
lea     EAX, [ESI+4]
sub     EBX, ESI
mov     [ESI-2], BX
add     ESI, EBX
dec     DWORD PTR [EBP-8]
loc_46162A8:
```
Nymaim’s DGA – Main Logic

- Test only the main logic, e.g. choose TLD
- Mock the rest!
- Might require several scenarios
Nymaim’s DGA – Main Logic

- Test only the main logic, e.g. choose TLD
- Mock the rest!
- Might require several scenarios

**Scenario:** Nymaim DGA chooses correct TLD from set of possible TLDs ["ru","net","in","com","xyz","info"]

**Given** the seeds

<table>
<thead>
<tr>
<th>seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>78670654</td>
</tr>
<tr>
<td>44370352</td>
</tr>
<tr>
<td>35461477</td>
</tr>
<tr>
<td>97912344</td>
</tr>
</tbody>
</table>

**When** DGA computes TLD

**Then** the TLD is ru
Nymaim’s DGA – PRNG (Xorshift)

- Next, we have a look at the PRNG (Xorshift)
- Still we do not want to deal with the seeds
- **Input**: five integers (4* seed + modulo)
- **Output**: integer [0, modulo - 1]
- Has side effects on the seeds!
Nymaim’s DGA – PRNG (Xorshift)

Scenario: PRNG works correctly for given seeds and modulo

Given the modulo 600
And the seeds

<table>
<thead>
<tr>
<th>seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>123172080</td>
</tr>
<tr>
<td>79962903</td>
</tr>
<tr>
<td>133504895</td>
</tr>
<tr>
<td>2326822159</td>
</tr>
</tbody>
</table>

When PRNG executes
Then the output is 1
Nymaim’s DGA – PRNG (Xorshift)

```assembly
xor    ecx, ecx
mov    eax, [ebp+arg_0]
or     eax, eax
setz   cl
or     eax, ecx
mov    esi, [ebp+arg_4]
imul   eax, 64h
or     eax, eax
jz     loc_46118AE
mov    edi, eax
mov    eax, [esi]
shl    eax, 08h
xor    eax, [esi]
mov    edx, [esi+4]
add    [esi], edx
mov    ecx, [esi+8]
add    [esi+4], ecx
mov    ebx, [esi+0Ch]
add    [esi+8], ebx
shr    ebx, 13h
xor    ebx, [esi+0Ch]
xor    ebx, eax
shr    eax, 8
xor    ebx, eax
mov    [esi+0Ch], ebx
mov    eax, ebx
add    eax, ecx
xor    edx, edx
div    edi
xchg   eax, edx
xor    edx, edx
mov    edi, 64h ; 'd'
div    edi
```
Nymaim’s DGA – PRNG (Xorshift)

```
xor    ecx, ecx
mov    eax, [ebp+arg_0]
or    eax, eax
setz   cl
or    eax, ecx
mov    esi, [ebp+arg_4]
imul   eax, 64h
or    eax, eax
jz    loc_46118AE
mov    edi, eax
mov    eax, [esi]
shr    eax, 08h
xor    eax, [esi]
mov    edx, [esi+4]
add    [esi], edx
mov    ecx, [esi+8]
add    [esi+4], ecx
mov    ebx, [esi+0Ch]
add    [esi+8], ebx
shr    ebx, 13h
xor    ebx, [esi+0Ch]
xor    ebx, eax
shr    eax, 8
xor    ebx, eax
mov    [esi+0Ch], ebx
mov    eax, ebx
add    eax, ecx
xor    edx, edx
div    edi
xchg   eax, edx
xor    edx, edx
mov    edi, 64h ; 'd'
div    edi
```
Nymaim’s DGA – Results

1 feature passed, 0 failed, 0 skipped
5 scenarios passed, 0 failed, 0 skipped
16 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.004s
Nymaim’s DGA – Results

1 feature passed, 0 failed, 0 skipped
5 scenarios passed, 0 failed, 0 skipped
16 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.004s
Nymaim’s DGA – Results

- Five tests of DGA’s features
- One end-to-end acceptance test
Nymaim’s DGA – Results

- Five tests of DGA’s features
- One end-to-end acceptance test
- Readable code
  - One class implementing the main logic
  - One class implementing the PRNG (strategy pattern)
  - One class serving as data structure
CONCLUSION & FUTURE WORK
Conclusion & Future Work

- BDD in malware analysis
- Case Study Nymaim
  - Check source code on *Bitbucket*!
- Future work
  - Automatic test case generation
  - Tools for gathering test data in RE context