

Recognition of binary patterns by Morphological Analysis

Aurélien Thierry (aurelien.thierry@inria.fr), Guillaume Bonfante, Joan Calvet, Jean-Yves Marion, Fabrice Sabatier

Recon 2012-06-16



Introduction

- Binary analysis

```
.text:00452C1B
.text:00452C1B      sub_452C1B      proc near          ; CODE XREF: sub
.text:00452C1B                                     ; sub_45317E+16j
.text:00452C1B
.text:00452C1B      var_h          = dword ptr -4
.text:00452C1B      arg_0          = dword ptr 4
.text:00452C1B      arg_h          = dword ptr 8
.text:00452C1B      arg_8          = dword ptr 0Ch
.text:00452C1B
.text:00452C1B      push         h
.text:00452C1B      pop          eax
.text:00452C1B      call         _alloca_probe
.text:00452C1E      push         ebx
.text:00452C23      push         ebp
.text:00452C24      push         esi
.text:00452C25      push         edi
.text:00452C26      mov          edi, [esp+14h+arg_0]
.text:00452C27      test         edi, edi
.text:00452C2B      jz           loc_453175
.text:00452C2D      mov          esi, [esp+14h+arg_8]
.text:00452C33      test        esi, esi
.text:00452C37      jz           loc_453175
.text:00452C39      mov          eax, [esp+14h+arg_h]
.text:00452C3F      mov          ecx, 80h
.text:00452C43      cmp          eax, ecx
.text:00452C48      jz           short loc_452C66
.text:00452C4A      cmp          eax, 0C0h
.text:00452C51      jz           short loc_452C62
.text:00452C53      cmp          eax, 100h
.text:00452C58      jz           short loc_452C62
.text:00452C5A      push         0FFFFFFEh
.text:00452C5C      pop          eax
.text:00452C5D      jmp          loc_453178
```

Introduction

- Binary analysis

```

.text:10021F60 public AES_set_encrypt_key
.text:10021F60 AES_set_encrypt_key proc near      ; CODE XREF: AES
.text:10021F60                                ; sub_1004978A+h
.text:10021F60
.text:10021F60 AES_set_encrypt (OpenSSL)
.text:10021F60 = duword ptr 0Ch
.text:10021F6A
.text:10021F6A push    h
.text:10021F6C pop     eax
.text:10021F6D call    _alloca_probe
.text:10021F72 push    ebx
.text:10021F73 push    ebp
.text:10021F74 push    esi
.text:10021F75 push    edi
.text:10021F76 mov     edi, [esp+14h+arg_0]
.text:10021F7A test    edi, edi
.text:10021F7C jz      loc_100224D1
.text:10021F82 mov     esi, [esp+14h+arg_8]
.text:10021F86 test    esi, esi
.text:10021F88 jz      loc_100224D1
.text:10021F8E mov     eax, [esp+14h+arg_4]
.text:10021F92 mov     ecx, 80h
.text:10021F97 cmp     eax, ecx
.text:10021F9B jz      short loc_10021FB5
.text:10021F9B cmp     eax, 0C0h
.text:10021FA0 jz      short loc_10021FB1
.text:10021FA2 cmp     eax, 100h
.text:10021FA7 jz      short loc_10021FB1
.text:10021FA9 push    0FFFFFFEh
.text:10021FAB pop     eax
.text:10021FAC jmp     loc_100224D4

```

- Identify libraries that do not need to be reversed

Introduction

- Binary analysis

```

.text:10021F6A public AES_set_encrypt_key
.text:10021F6A AES_set_encrypt key proc near ; CODE XREF: AES
.text:10021F6A ; sub_1004970A+4
.text:10021F6A
.text:10021F6A
.text:10021F6A
.text:10021F6A AES_set_encrypt (OpenSSL)
.text:10021F6A = dword ptr 0Ch
.text:10021F6A
.text:10021F6A push 4
.text:10021F6C pop eax
.text:10021F6D call alloca_probe
.text:10021F72 push ebx
.text:10021F73 push ebp
.text:10021F74 push esi
.text:10021F75 push edi
.text:10021F76 mov edi, [esp+14h+arg_0]
.text:10021F7A test edi, edi
.text:10021F7C jz loc_100224D1
.text:10021F82 mov esi, [esp+14h+arg_8]
.text:10021F86 test esi, esi
.text:10021F88 jz loc_100224D1
.text:10021F8E mov eax, [esp+14h+arg_4]
.text:10021F92 mov ecx, 00h
.text:10021F97 cmp eax, ecx
.text:10021F99 jz short loc_10021FB5
.text:10021F9B cmp eax, 0C0h
.text:10021FA0 jz short loc_10021FB1
.text:10021FA2 cmp eax, 100h
.text:10021FA7 jz short loc_10021FB1
.text:10021FA9 push 0FFFFFFFh
.text:10021FAB pop eax
.text:10021FAC jmp loc_100224D4

```

- Identify libraries that do not need to be reversed

Our approach :

- Control flow graph comparison
- Import results in IDA

Waledac malware and OpenSSL

- Spamming botnet
- Use of cryptography for communication : RSA and AES

Waledac malware and OpenSSL

- Spamming botnet
- Use of cryptography for communication : RSA and AES

```
aurelien:~/R$ strings Waledac\ v48\ unpacked.int | grep OpenSSL  
EC part of OpenSSL 0.9.8e 23 Feb 2007  
ECDSA part of OpenSSL 0.9.8e 23 Feb 2007
```

- OpenSSL 0.9.8e (Feb 2007) used for cryptography

Waledac malware and OpenSSL

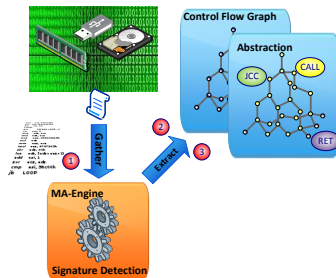
- Spamming botnet
- Use of cryptography for communication : RSA and AES

```
aurelien:~/R$ strings Waledac\ v48\ unpacked.int | grep OpenSSL  
EC part of OpenSSL 0.9.8e 23 Feb 2007  
ECDSA part of OpenSSL 0.9.8e 23 Feb 2007
```

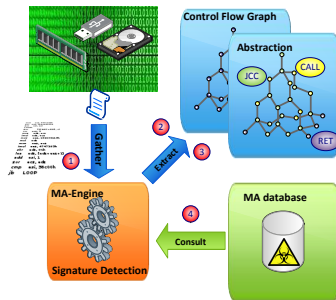
- OpenSSL 0.9.8e (Feb 2007) used for cryptography
- Which functions are specifically used ?

Morphological Analysis : Scanning a file

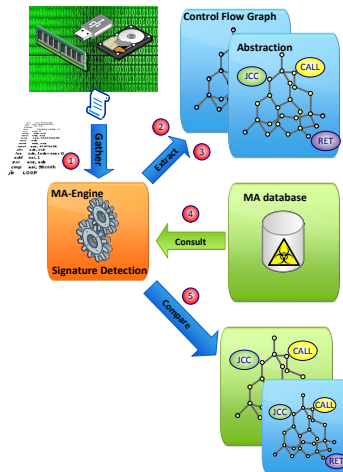
Step 2 : Scan



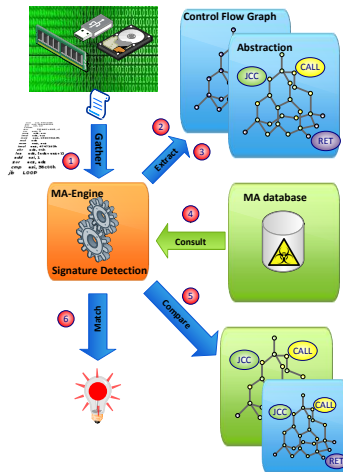
Step 2 : Scan



Step 2 : Scan

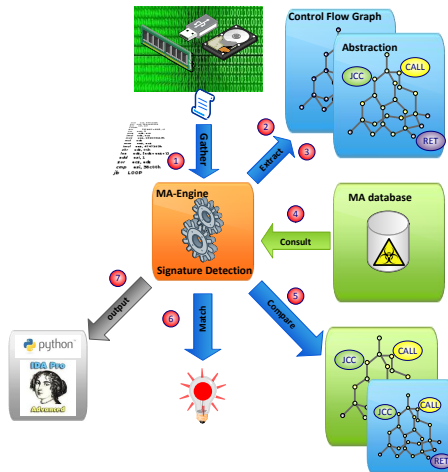


Step 2 : Scan



Morphological Analysis : Scanning a file

Step 2 : Scan



Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution

Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution

ASM code

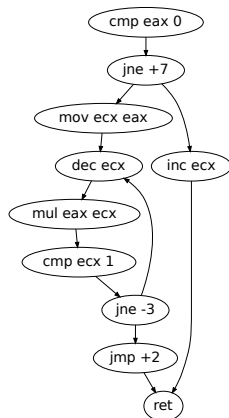
```
cmp eax 0
jne +7
mov ecx eax
dec ecx
mul eax ecx
cmp ecx 1
jne -3
jmp +2
inc ecx
ret
```

Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution

ASM code

```
cmp eax 0
jne +7
mov ecx eax
dec ecx
mul eax ecx
cmp ecx 1
jne -3
jmp +2
inc ecx
ret
```

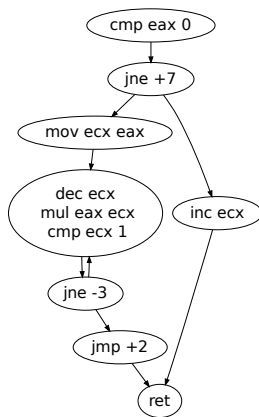
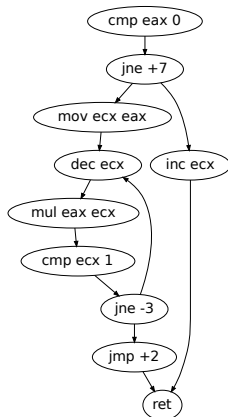


Control flow graph recovery

Control Flow Graph (CFG) : oriented graph in which nodes are instruction addresses and edges represent all paths that might be traversed during execution

ASM code

```
cmp eax 0
jne +7
mov ecx eax
dec ecx
mul eax ecx
cmp ecx 1
jne -3
jmp +2
inc ecx
ret
```



Control flow graph recovery

Extraction of the control flow graph from a binary :

- Static analysis from entrypoints when possible (BeaEngine)
- Dynamic analysis otherwise (Intel's Pintools)

Control flow graph recovery

Extraction of the control flow graph from a binary :

- Static analysis from entrypoints when possible (BeaEngine)
- Dynamic analysis otherwise (Intel's Pintools)

Nodes of the control flow graph :

- Sequential instructions do not modify the control flow
- 4 types of instructions have an impact on the CFG (*jmp*, *call*, *jcc*, et *ret*)

Control flow graph construction & reduction

N _{th} instruction	Control flow graph
Sequential instruction	 <pre> graph TD INST((INST)) --> N1((N+1)) </pre>
jmp K	 <pre> graph TD JMP((JMP)) --> K((K)) </pre>
call K	 <pre> graph TD CALL((CALL)) --> N1((N+1)) CALL --> K((K)) </pre>
jcc K	 <pre> graph TD JCC((JCC)) --> N1((N+1)) JCC --> K((K)) </pre>
ret	 <pre> graph TD RET((RET)) </pre>

Control flow graph construction & reduction

N_{th} instruction	Control flow graph
Sequential instruction	<pre> graph TD INST((INST)) --> N1((N+1)) </pre>
jmp K	<pre> graph TD JMP((JMP)) --> K((K)) </pre>
call K	<pre> graph TD CALL((CALL)) --> N1((N+1)) CALL --> K((K)) </pre>
jcc K	<pre> graph TD JCC((JCC)) --> N1((N+1)) JCC --> K((K)) </pre>
ret	<pre> graph TD RET((RET)) </pre>

Remove sequential instructions	<pre> graph TD subgraph Before N1((N)) --> INST1((INST)) INST1 --> N2((N+2)) end subgraph After N2((N)) --> N3((N+2)) end Before --> After </pre>
Realign jumps	<pre> graph TD subgraph Before N1((N)) --> JMP1((JMP)) JMP1 --> K1((K)) end subgraph After N2((N)) --> K2((K)) end Before --> After </pre>
Remove false calls	<pre> graph TD subgraph Before N1((N)) --> CALL1((CALL)) CALL1 --> N2((N+2)) CALL1 --> RET1((RET)) N2 --> N3((N+3)) end subgraph After N2((N)) --> N3((N+2)) N3 --> N4((N+3)) end Before --> After </pre>

Reduction of the control flow graph

The CFG is reduced :

- Reduce the size of the graph (fastens the algorithms)
- More abstract form : detect slight changes (junk code insertion, code re-ordering)

Reduction of the control flow graph

The CFG is reduced :

- Reduce the size of the graph (fastens the algorithms)
- More abstract form : detect slight changes (junk code insertion, code re-ordering)

Waledac with static analysis :

- 38236 nodes before reduction
- 14626 nodes after reduction

Reduction on Waledac

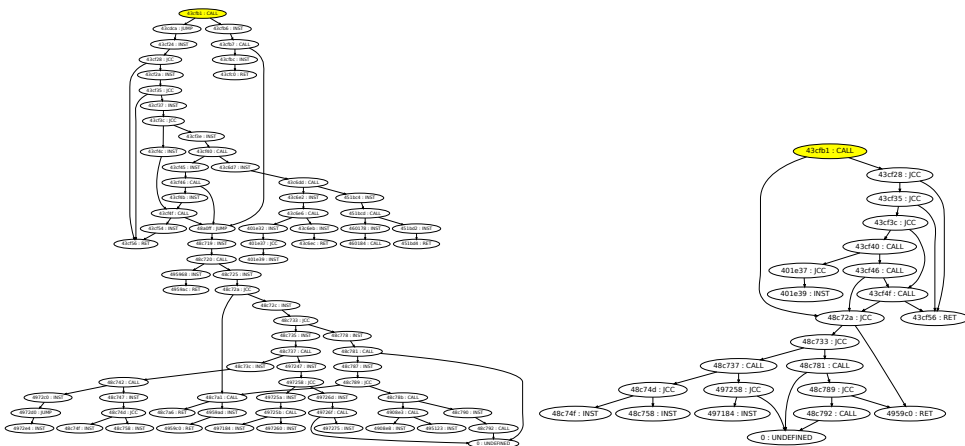
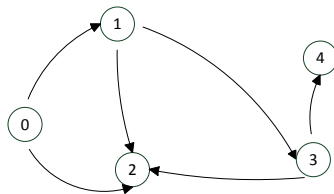
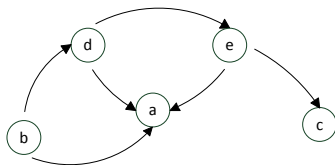


Figure: Part of Waledac without reduction (80 nodes) and with reduction (23 nodes)

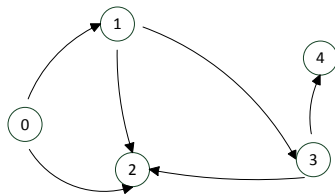
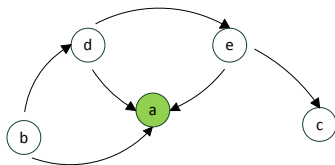
Graph matching

Graph isomorphism detection



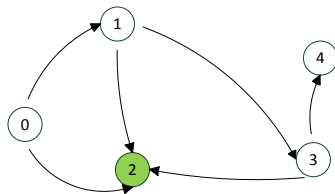
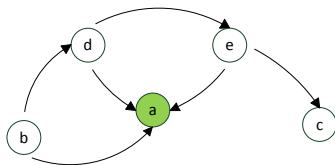
Graph matching

Graph isomorphism detection



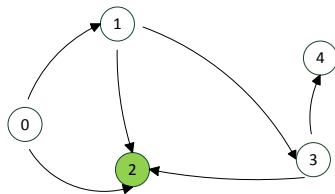
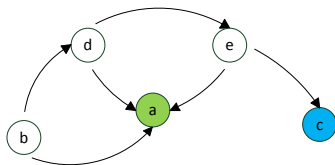
Graph matching

Graph isomorphism detection



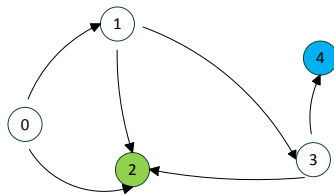
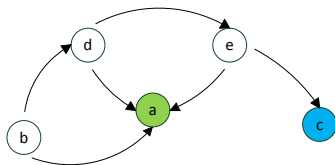
Graph matching

Graph isomorphism detection



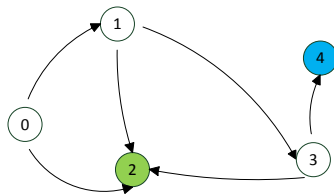
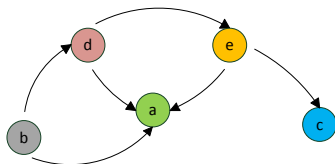
Graph matching

Graph isomorphism detection



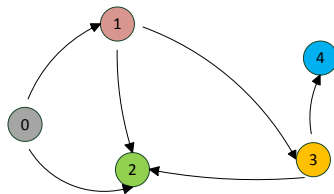
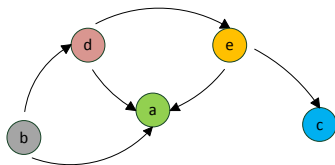
Graph matching

Graph isomorphism detection



Graph matching

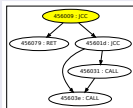
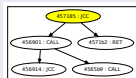
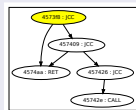
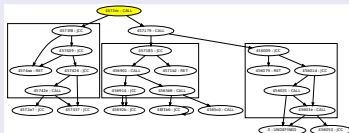
Graph isomorphism detection



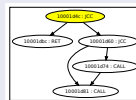
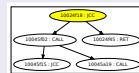
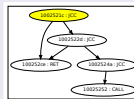
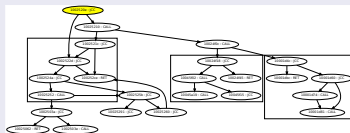
- Entire graphs are not isomorphic

[illegible]

Waledac



OpenSSL



Subgraphs

- Both graphs are cut into many small subgraphs
- Generated through BFS (Breadth First Search) from each nodes

Subgraphs

- Both graphs are cut into many small subgraphs
- Generated through BFS (Breadth First Search) from each nodes
- Their size is limited (typically 24 nodes)
- Search graph isomorphisms between subgraphs of both binaries

Graph isomorphism problem

- Graph isomorphism has no solution in polynomial time in the general case
- The problem is in NP
- General solutions are slow

Graph isomorphism problem

- Graph isomorphism has no solution in polynomial time in the general case
- The problem is in NP
- General solutions are slow

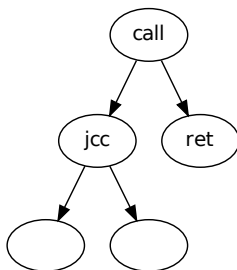
Property (Simplification)

Our subgraphs :

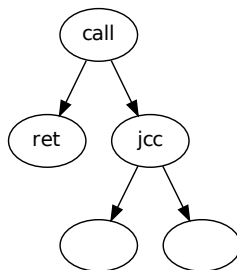
- *Have a root node (from which every other node is reachable)*
 - *Each node has at most 2 children (call or jcc)*
 - *Children are ordered*
- This problem is in P

Graph isomorphism problem

- Does not exactly resolve the graph isomorphism problem
- But there are fast solutions (polynomial time)



(v) Original graph



(w) Undetected graph

Morphological analysis engine

- Signatures are subgraphs from reduced control flow graphs
- Obtained statically or dynamically
- A database (tree automata) is filled with the signatures

Morphological analysis engine

- Signatures are subgraphs from reduced control flow graphs
- Obtained statically or dynamically
- A database (tree automata) is filled with the signatures
- Learning and scanning is fast (Intel Core i5 CPU M560 2.67GHz)

Operation	Files	Time (s)
Learn	44 binaries (< 2000 nodes)	1.2s
Scan	44 binaries (< 2000 nodes)	1.1s
Learn	OpenSSL (28313 nodes)	12s
Scan	Waledac (14626 nodes)	2.0s

Compare Waledac and OpenSSL

- Waledac uses OpenSSL 0.9.8e (Feb 2007)
- OpenSSL learnt with reduction

Compare Waledac and OpenSSL

- Waledac uses OpenSSL 0.9.8e (Feb 2007)
- OpenSSL learnt with reduction
- One DLL is matched (libeay.dll)

OpenSSL version	Comment	Results (common subgraphs)
0.9.8x	Released in May 2012	53
0.9.8e	Compiled for performance (/0x /02)	53
0.9.8e	Compiled for file size (/01)	1264

Compare Waledac and OpenSSL

- Compile OpenSSL 0.9.8e with option /O1 (size optimization)
- 1264 common subgraphs between one of the DLLs (libeay.dll) and Waledac !!
- We want to know which functions are matched
- We will compare the matched code of OpenSSL and Waledac

Code and nodes

- The larger the matched subgraphs are, the more accurate the matching

Code and nodes

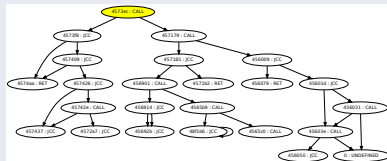
- The larger the matched subgraphs are, the more accurate the matching
- Learns and scans with increasing number of nodes from 24
- Associate nodes that are in the largest subgraphs

Code and nodes

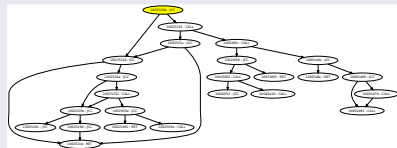
- The larger the matched subgraphs are, the more accurate the matching
- Learns and scans with increasing number of nodes from 24
- Associate nodes that are in the largest subgraphs
- Outputs matched nodes for each size for IDA

Code and nodes

Waledac



OpenSSL



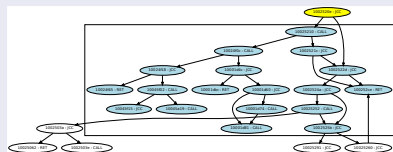
Code and nodes

- Greatest subgraph found has 18 nodes
- Corresponding nodes in matched subgraphs are associated
- Then associate free nodes on matching subgraphs of lesser size

Waledac



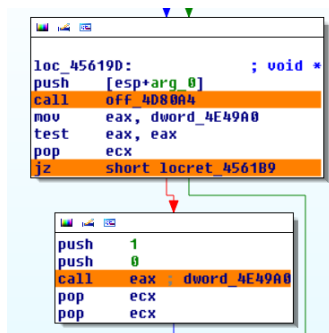
OpenSSL



IDA plugin

With both binaries opened in IDA

- Imports the list of matched nodes
- Marks them in IDA
- Provides browsing through corresponding nodes in both instances



Waledac / OpenSSL : common subroutines

libeay32-098e.dll	subroutine	Waledac48.int	subroutine
10002CDE	CRYPTO_new_ex_data	00455B5C	sub_455B55
10002CE0	CRYPTO_new_ex_data	00455B5E	sub_455B55
10002D0A	CRYPTO_free_ex_data	00455B72	sub_455B6B
10002D0C	CRYPTO_free_ex_data	00455B74	sub_455B6B
10021F6D	AES_set_encrypt_key	00452C1E	sub_452C1B
10021F7C	AES_set_encrypt_key	00452C2D	sub_452C1B
10021F88	AES_set_encrypt_key	00452C39	sub_452C1B
10021F99	AES_set_encrypt_key	00452C4A	sub_452C1B
10021FA0	AES_set_encrypt_key	00452C51	sub_452C1B
10021FA7	AES_set_encrypt_key	00452C58	sub_452C1B
10021FB3	AES_set_encrypt_key	00452C64	sub_452C1B
10021FD9	AES_set_encrypt_key	00452C8A	sub_452C1B
10021FEF	AES_set_encrypt_key	00452CA0	sub_452C1B
100224D9	AES_set_encrypt_key	0045317D	sub_452C1B
100224E0	AES_set_decrypt_key	00453184	sub_45317E
100224F0	AES_set_decrypt_key	00453194	sub_45317E
100224FA	AES_set_decrypt_key	0045319E	sub_45317E

Figure: Matching nodes are in corresponding subroutines

Waledac / OpenSSL : common subroutines

- AES : AES_set_encrypt_key, AES_set_decrypt_key
- X509 : X509_PUBKEY_set, X509_PUBKEY_get
- RSA / DSA : RSA_free, DSA_size, DSA_new_method
- BN (Big Number lib) : BN_is_prime_fasttest_ex, BN_ctx_new, BN_mod_inverse
- CRYPTO : CRYPTO_lock, CRYPTO_malloc
- Misc OpenSSL routines : UI, encoding...
- ...

Comparing matched code : AES_set_encrypt_key

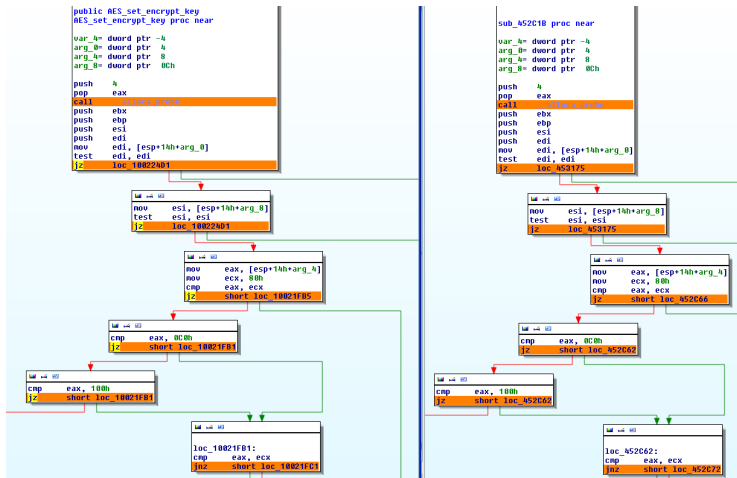
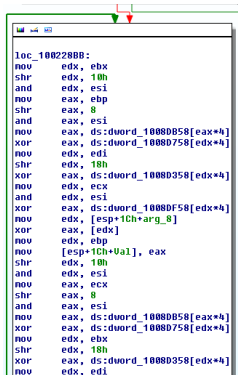


Figure: Matched code between OpenSSL (left) and Waledac (right)

Comparing code : AES_encrypt

- Not detected

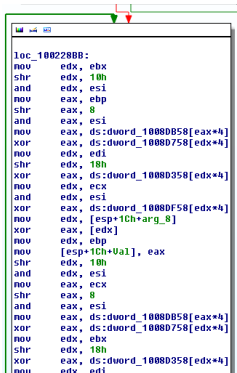


```
loc_1002280B:
mov     edx, ebx
shr     edx, 10h
and     edx, esi
mov     eax, ebp
shr     eax, 8
and     eax, esi
mov     eax, ds:dword_10080B58[eax*4]
xor     eax, ds:dword_10080758[edx*4]
mov     edx, edi
shr     edx, 18h
xor     eax, ds:dword_10080358[edx*4]
mov     ecx, edx
and     ecx, esi
xor     eax, ds:dword_10080F58[edx*4]
mov     edx, [esp+1Ch+arg_8]
xor     eax, [edx]
mov     edx, ebp
mov     [esp+1Ch+0A1], eax
shr     edx, 10h
and     edx, esi
mov     ecx, eax
shr     ecx, 8
and     ecx, esi
mov     eax, ds:dword_10080B58[eax*4]
xor     eax, ds:dword_10080758[edx*4]
mov     edx, ebx
shr     edx, 18h
xor     eax, ds:dword_10080358[edx*4]
mov     edi, eax
```

Figure: AES_encrypt subroutine

Comparing code : AES_encrypt

- Not detected
- Control flow graph too small



```
loc_1002280B:  
mov     edx, ebx  
shr     edx, 10h  
and     edx, esi  
mov     eax, ebp  
shr     eax, 8  
and     eax, esi  
mov     eax, ds:dword_1008D858[eax*4]  
xor     eax, ds:dword_1008D758[edx*4]  
mov     edx, edi  
shr     edx, 18h  
xor     eax, ds:dword_1008D358[edx*4]  
mov     edx, ecx  
and     edx, esi  
xor     eax, ds:dword_1008DF58[edx*4]  
mov     edx, [esp+1Ch+arg_8]  
xor     eax, [edx]  
mov     edx, ebp  
mov     [esp+1Ch+0A1], eax  
shr     edx, 10h  
and     edx, esi  
mov     eax, ecx  
shr     eax, 8  
and     eax, esi  
mov     eax, ds:dword_1008D858[eax*4]  
xor     eax, ds:dword_1008D758[edx*4]  
mov     edx, ebx  
shr     edx, 18h  
xor     eax, ds:dword_1008D358[edx*4]  
mov     edx, edi
```

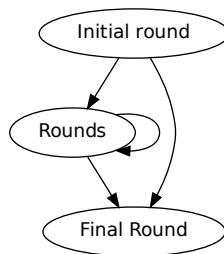


Figure: AES_encrypt subroutine

Figure: Simplified AES_encrypt CFG

Waledac / OpenSSL : Findings

- OpenSSL 0.9.8e compiled for being small (option /O1)

Waledac / OpenSSL : Findings

- OpenSSL 0.9.8e compiled for being small (option /O1)
- Use of AES for symmetric encryption
- X.509 (certificate) handling, use of RSA and/or DSA algorithm
- Calls to primality tests (consistent with asymmetric encryption like RSA but not exclusively)

Waledac / OpenSSL : Findings

- OpenSSL 0.9.8e compiled for being small (option /O1)
- Use of AES for symmetric encryption
- X.509 (certificate) handling, use of RSA and/or DSA algorithm
- Calls to primality tests (consistent with asymmetric encryption like RSA but not exclusively)
- Waledac actually uses X509/RSA and AES encryption
- We were able to find out without actually reversing its code

Duqu and Stuxnet

- Static analysis on their decrypted (and unpacked) main DLLs (maindll.dll for Stuxnet and netp191.pnf for Duqu)

Duqu and Stuxnet

- Static analysis on their decrypted (and unpacked) main DLLs (maindll.dll for Stuxnet and netp191.pnf for Duqu)

First analysis :

- 26.5% of Duqu's subgraphs are common with Stuxnet (846 subgraphs)
- 60.3% of Duqu's nodes are in subgraphs matching with Stuxnet (2215 nodes)
- Duqu and Stuxnet are strongly related

Duqu / Stuxnet : common subroutines

maindll.decrypted.u	subroutine	netp191_Decrypter	subroutine
10042DB0	sub_10042CD2	100136DB	sub_100135FD
10042DC0	sub_10042CD2	100136EB	sub_100135FD
10042DC5	sub_10042CD2	100136F0	sub_100135FD
10042DD3	sub_10042CD2	100136FE	sub_100135FD
10042DE0	sub_10042CD2	1001370B	sub_100135FD
10043116	sub_100430CE	1001353F	sub_100134F7
1004311F	sub_100430CE	10013548	sub_100134F7
10043138	sub_100430CE	10013561	sub_100134F7
1004314D	sub_100430CE	10013576	sub_100134F7
10043155	sub_100430CE	1001357E	sub_100134F7
10043157	sub_100430CE	10013580	sub_100134F7
10043161	sub_100430CE	1001358A	sub_100134F7
1004317B	sub_100430CE	100135A4	sub_100134F7
10043183	sub_1004317C	100144E3	sub_100144DC
1004318D	sub_1004317C	100144ED	sub_100144DC

Duqu / Stuxnet : subroutine identification

- Some of the common subroutines come from standard libraries (libc...)
- They are documented and should not be manually reversed

Duqu / Stuxnet : subroutine identification

- Some of the common subroutines come from standard librairies (libc...)
- They are documented and should not be manually reversed
- msvcrt80.dll : Microsoft Visual C++ Run-Time
- How to identify its code within Duqu / Stuxnet in IDA ?

Duqu / Stuxnet : libc identification

- Learn msvcrt80.dll ('libc') and scan Duqu, Stuxnet

Duqu / Stuxnet : libc identification

- Learn msvcrt80.dll ('libc') and scan Duqu, Stuxnet

IDA plugin will :

- Mark the nodes common with msvcrt80.dll
- Rename the matched subroutines

Duqu / Stuxnet : common subroutines

maindll.decrypted.	subroutine	netp191_Decrypt	subroutine
10042DB0	sub_10042CD2	100136DB	sub_100135FD
10042DC0	sub_10042CD2	100136EB	sub_100135FD
10042DC5	sub_10042CD2	100136F0	sub_100135FD
10042DD3	sub_10042CD2	100136FE	sub_100135FD
10042DE0	sub_10042CD2	1001370B	sub_100135FD
10043116	msvcr80\$__beginthreadex	1001353F	msvcr80\$__beginthreadex
1004311F	msvcr80\$__beginthreadex	10013548	msvcr80\$__beginthreadex
10043138	msvcr80\$__beginthreadex	10013561	msvcr80\$__beginthreadex
1004314D	msvcr80\$__beginthreadex	10013576	msvcr80\$__beginthreadex
10043155	msvcr80\$__beginthreadex	1001357E	msvcr80\$__beginthreadex
10043157	msvcr80\$__beginthreadex	10013580	msvcr80\$__beginthreadex
10043161	msvcr80\$__beginthreadex	1001358A	msvcr80\$__beginthreadex
1004317B	msvcr80\$__beginthreadex	100135A4	msvcr80\$__beginthreadex
10043183	msvcr80\$__free	100144E3	msvcr80\$__free
1004318D	msvcr80\$__free	100144ED	msvcr80\$__free

Figure: Renamed subroutines matching between Duqu and Stuxnet

Highlighting msvcrt80.dll in Stuxnet

```

msvcrt80$__m__initlocknum
msvcrt80$__lock
msvcrt80$__sbh_find_block
msvcrt80$__sbh_free_block
msvcrt80$__sbh_alloc_new
msvcrt80$__sbh_alloc_new
msvcrt80$__sbh_resize_block
msvcrt80$__sbh_alloc_block
sub_1004720F
sub_10047248
sub_1004725B
sub_10047293
sub_100472A4
nullsub_2
sub_100473C0
sub_100473EC
sub_10047441
sub_1004745D
sub_100474E5
msvcrt80$__wchartodigit
sub_10047730
sub_10047770
msvcrt80$fastcopy_I
msvcrt80$VEC memcpy
0047E31 ; ===== S U B R O U T I N E =====
0047E31
0047E31 ; Function found in msvcrt80
0047E31 ; Attributes: bp-based frame
0047E31
0047E31 ; int __cdecl msvcrt80$realloc(LPVOID lpMem, int)
0047E31 public msvcrt80$realloc
0047E31 msvcrt80$realloc proc near ; CODE XREF: sub_10044C90
0047E31
0047E31 var_20 = dword ptr -20h
0047E31 var_1C = dword ptr -1Ch
0047E31 ms_exc = CPPEH_RECORD ptr -18h
0047E31 lpMem = dword ptr 8
0047E31 arg_4 = dword ptr 0Ch
0047E31
0047E31 ; FUNCTION CHUNK AT .text:10047F7E SIZE 000000CE BYTES
0047E31
0047E31 push 10h
0047E31 push offset stru_1006B278
0047E38 call SEH_prolog4
0047E3D mov ebx, [ebp+lpMem]
0047E40 test ebx, ebx
0047E42 jnz short loc_10047E52
0047E44 push [ebp+arg_4]
0047E47 call sub_10043259
0047E4C pop ecx
0047E4D jmp loc_1004801E

```

Figure: Colored (yellow) code of msvcrt80.dll in Stuxnet, subroutines are renamed

Duqu / Stuxnet : summary

- From the decrypted and unpacked DLLs from Stuxnet, we are able to automatically find code shared with Duqu
- Before reversing, we identify standard (msvcr80.dll) subroutines
- With IDA, we can identify and browse matching subroutines

Conclusion

- Identify used librairies
- Show code similarities
- IDA UI for browsing matched code

Conclusion

- Identify used librairies
- Show code similarities
- IDA UI for browsing matched code
- Thank you
- Any question ? (aurelien.thierry@inria.fr)