Table of Contents

Basic Static Analysis

Purpose

The idea here is to glean any information from the file itself. This information is useful because it gives us an idea of what to expect when we run the executable, and may give us enough information to tweak our tools to get more out of the following basic dynamic analysis.

File information

```
File: assignment3.exe
MD5:  989258474585d26352baac6fbff91f0e
Size: 67072
```

VirusTotal

Detection ratio: 12/47
Packers Identified: YodaProt, UPX
TimeStamp: 2001:08:17 21:54:24+01:00
Interesting Sections:

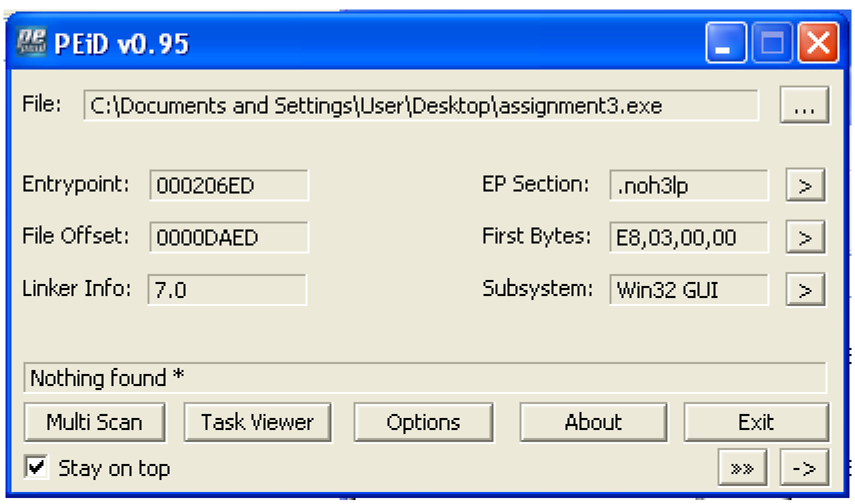
Name	Virtual address	Virtual size	Raw size	Entropy	MD5
.x01	102400	28672	10752	7.98	be75e38e070a3ab7c10b26242a233d21
.noh3lp	131072	36864	12800	7.96	7f6fe4953a42d19df2e2c62d9917005c

Figure 1: Section information from VirusTotal

These are non-standard section names and they have high entropy values.

PEiD

PEiD did not identify a packer, and searching for noh3lp on google did not return any results.



Dependency Walker

KERNEL32.DLL
- LoadLibraryA
- GetProcAddress

Strings

Some function-like names that aren't in the imports list.

```
000040EB  StartupInfoA
000040F8  ModuleHand
0000410E  lstr
0000411A  LocalFree
…
```

Verison info, this is our hint to this exe being a packed solitaire.

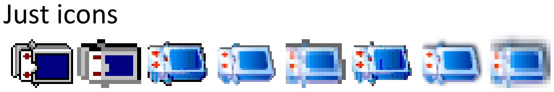
```
0000A6A4  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
0000A6DD  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
0000A728  <assemblyIdentity type="win32" name="Microsoft.Windows.Accessories.Games.Solitaire" version="1.0.0.0"
processorArchitecture="x86"/>
0000A7AD  <description>Solitaire Game</description>
0000A7D8  <dependency>
0000A7E6      <dependentAssembly>
0000A7FF          <assemblyIdentity
0000A81A              type="win32"
0000A834              name="Microsoft.Windows.Common-Controls"
0000A86A              version="6.0.0.0"
0000A889              language="*"
0000A8A3              publicKeyToken="6595b64144ccf1df"
0000A8D2              processorArchitecture="x86"/>
0000A8FD      </dependentAssembly>
0000A917  </dependency>
0000A926  </assembly>
…
```

These strings share a pattern, so they stuck out to me. They may be relevant during the unpacking process.

```
00010257  $#"!
0001025F  ,+*)('&%
000102E7  tsrqponm|{zyxwvu
00010307  TSRQPONM\[ZYXWVUdcba`_^]lkjihgfe43210/.-<;:98765DCBA@?>=LKJIHGFE
00010357  $#"!
0001035F  ,+*)('&%
000103E7  tsrqponm|{zyxwvu
00010407  TSRQPONM\[ZYXWVUdcba`_^]lkjihgfe43210/.-<;:98765DCBA@?>=LKJIHGFE
00010457  $#"!
0001045F  ,+*)('&%
000104BE  M'&%#3p
00010542  tsrqponmy
0001054E  xwvu
```

Resource Hacker

Just icons



Conclusion

Due to the lack of strings and imports, the section names, and virus total's report, it is very likely that this executable is packed at least once. The version info string section and the icons lead us to believe that this could be a packed version of solitaire.

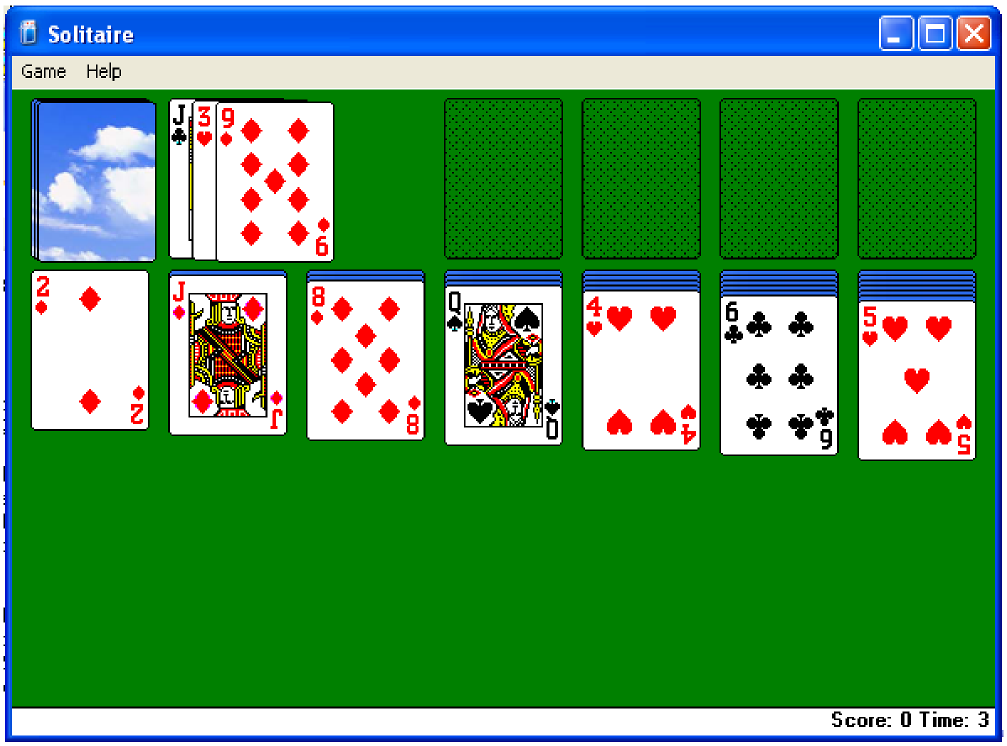
Basic Dynamic Analysis

Purpose

By running the program and analyzing its effects on the system we can get a first-hand look of the malware in action. This gives us some context when we disassemble the executable during advanced static analysis. It also allows us to develop host or network based signatures to identify the presence of this malware on a system in the future.

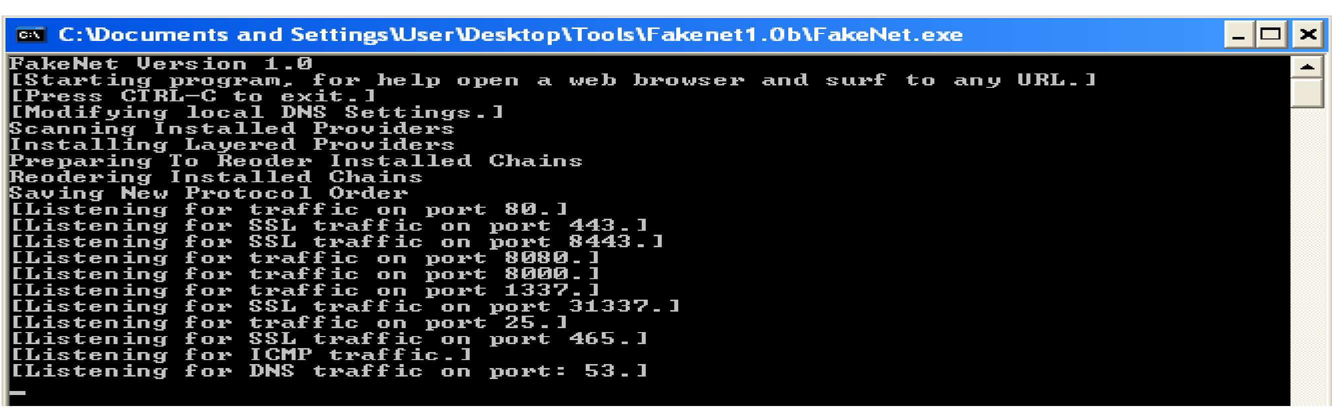
Program

It plays like a normal solitaire game, and doesn't have any apparent differences.



FakeNet

There was no common network traffic.

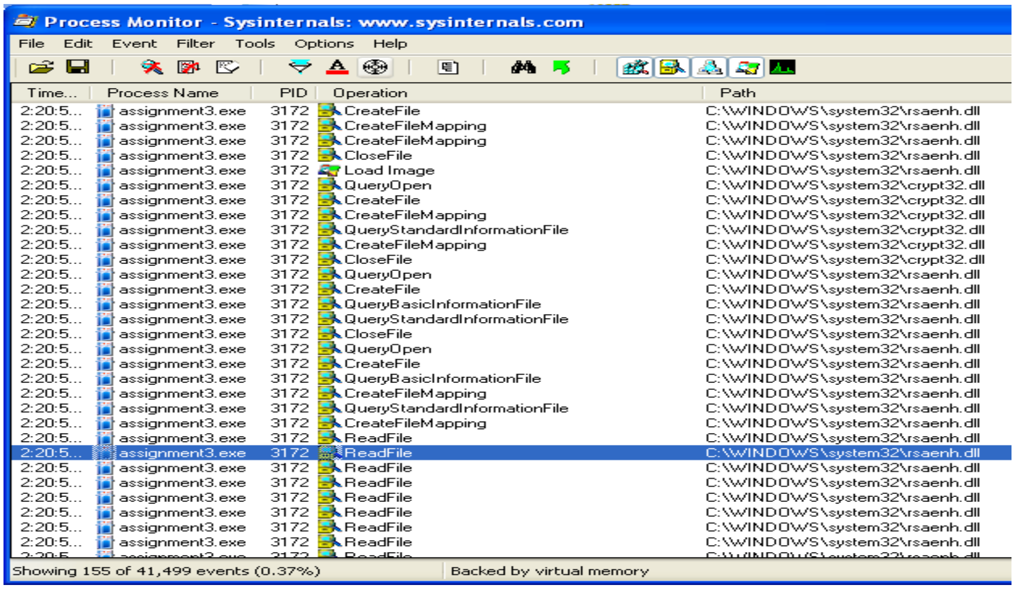


RegShot

There is nothing suspicious detected by RegShot.

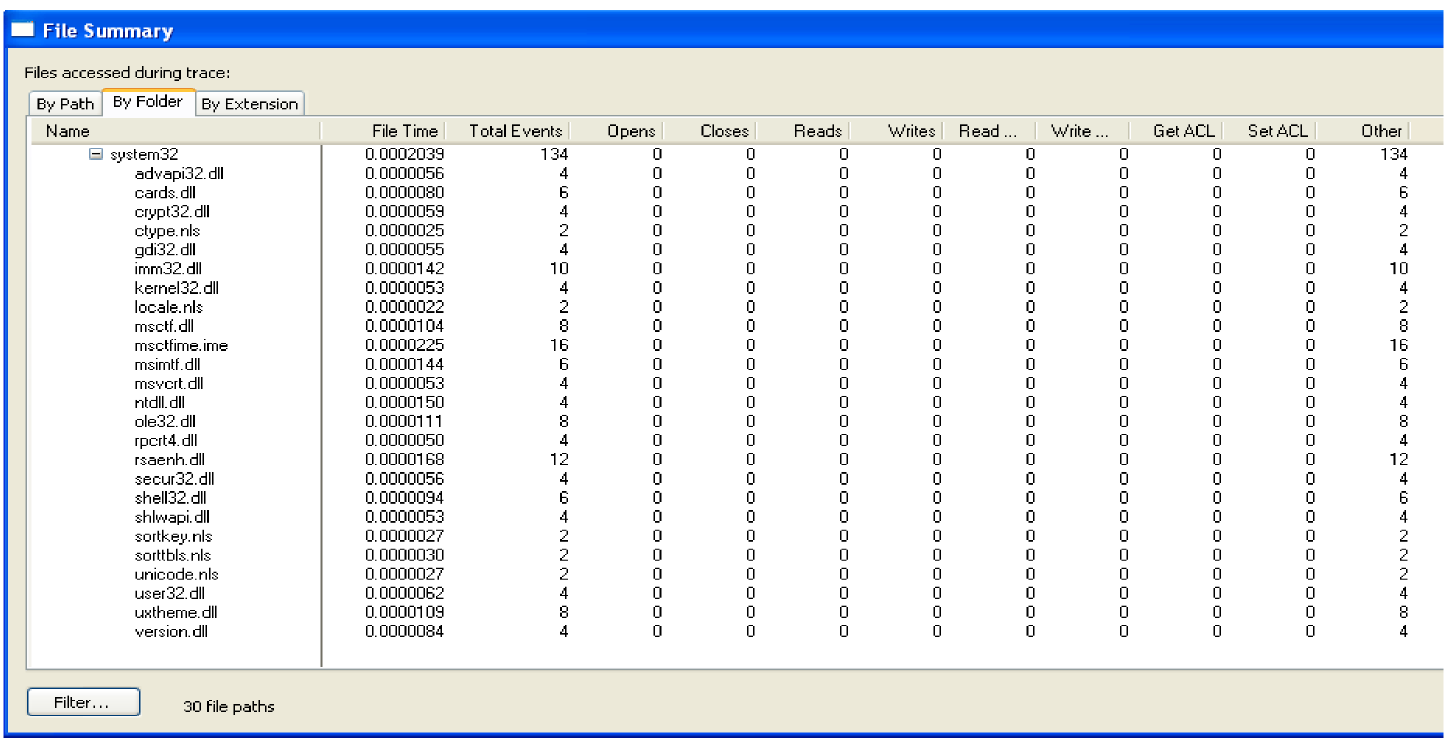
ProcessMonitor

Access crypto libs a lot, probably for the unpacking procedure.



Time...	Process Name	PID	Operation	Path
2:20:5...	assignment3.exe	3172	CreateFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CreateFileMapping	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CreateFileMapping	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CloseFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	Load Image	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	QueryOpen	C:\WINDOWS\system32\crypt32.dll
2:20:5...	assignment3.exe	3172	CreateFile	C:\WINDOWS\system32\crypt32.dll
2:20:5...	assignment3.exe	3172	CreateFileMapping	C:\WINDOWS\system32\crypt32.dll
2:20:5...	assignment3.exe	3172	QueryStandardInformationFile	C:\WINDOWS\system32\crypt32.dll
2:20:5...	assignment3.exe	3172	CreateFileMapping	C:\WINDOWS\system32\crypt32.dll
2:20:5...	assignment3.exe	3172	CloseFile	C:\WINDOWS\system32\crypt32.dll
2:20:5...	assignment3.exe	3172	QueryOpen	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CreateFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	QueryBasicInformationFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	QueryStandardInformationFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CloseFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	QueryOpen	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CreateFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	QueryBasicInformationFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CreateFileMapping	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	QueryStandardInformationFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	CreateFileMapping	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll
2:20:5...	assignment3.exe	3172	ReadFile	C:\WINDOWS\system32\rsaenh.dll

We can also see it load the modules it needs now that it is unpacked.



Name	File Time	Total Events	Opens	Closes	Reads	Writes	Read ...	Write ...	Get ACL	Set ACL	Other
system32	0.0002039	134	0	0	0	0	0	0	0	0	134
advapi32.dll	0.0000056	4	0	0	0	0	0	0	0	0	4
cards.dll	0.0000080	6	0	0	0	0	0	0	0	0	6
crypt32.dll	0.0000059	4	0	0	0	0	0	0	0	0	4
ctype.nls	0.0000025	2	0	0	0	0	0	0	0	0	2
gdi32.dll	0.0000055	4	0	0	0	0	0	0	0	0	4
imm32.dll	0.0000142	10	0	0	0	0	0	0	0	0	10
kernel32.dll	0.0000053	4	0	0	0	0	0	0	0	0	4
locale.nls	0.0000022	2	0	0	0	0	0	0	0	0	2
msctf.dll	0.0000104	8	0	0	0	0	0	0	0	0	8
msctfime.ime	0.0000225	16	0	0	0	0	0	0	0	0	16
msimtf.dll	0.0000144	6	0	0	0	0	0	0	0	0	6
msvcrt.dll	0.0000053	4	0	0	0	0	0	0	0	0	4
ntdll.dll	0.0000150	4	0	0	0	0	0	0	0	0	4
ole32.dll	0.0000111	8	0	0	0	0	0	0	0	0	8
rpcrt4.dll	0.0000050	4	0	0	0	0	0	0	0	0	4
rsaenh.dll	0.0000168	12	0	0	0	0	0	0	0	0	12
secur32.dll	0.0000056	4	0	0	0	0	0	0	0	0	4
shell32.dll	0.0000094	6	0	0	0	0	0	0	0	0	6
shlwapi.dll	0.0000053	4	0	0	0	0	0	0	0	0	4
sortkey.nls	0.0000027	2	0	0	0	0	0	0	0	0	2
sorttbls.nls	0.0000030	2	0	0	0	0	0	0	0	0	2
unicode.nls	0.0000027	2	0	0	0	0	0	0	0	0	2
user32.dll	0.0000062	4	0	0	0	0	0	0	0	0	4
uxtheme.dll	0.0000109	8	0	0	0	0	0	0	0	0	8
version.dll	0.0000084	4	0	0	0	0	0	0	0	0	4

ProcessExplorer

Now that the process is unpacked in memory, we can use ProcessExplorer to dump the new strings.
We see a lot of Solitaire related strings, followed by

    CLSID\{ADB880A6-D8FF-11CF-9377-00AA003B7A11}\InprocServer32

some license info, and all of the imports. The same meta-data tag from before is still there, along with the original imports.
This is also found at the end.

    Microsoft Base Cryptographic Provider v1.0

Conclusion

Our basic static analysis confirmed our original belief that this program is a packed solitaire. We did not see any hints of malicious activity, though it is possible that the program is using anti-VM techniques to avoid detection. We will be able to spot these during advanced static analysis.

Unpacking

Purpose

Since this program is packed we will need to unpack it before we can do anything else useful.

Analysis of the Packers

Recall that virus total detected the packers YodaProt and UPX. It can be useful to research the packers for tutorials, auto-unpackers, and tips. A lot of the time packers use the same technique between versions, if you know what to look for you can save a lot of time. Since it doesn't look like UPX is the first packer, we will tackle YodaProt first.

YodaProt

I googled around for a bit but I couldn't find any information on y0da's packer/protector. This is unfortunate, because it means I will have to tackle the program manually.

UPX

From previous experience, UPX can be unpacked automatically with the command-line tool. If you have to unpack it manually just look for a tail jmp / call.

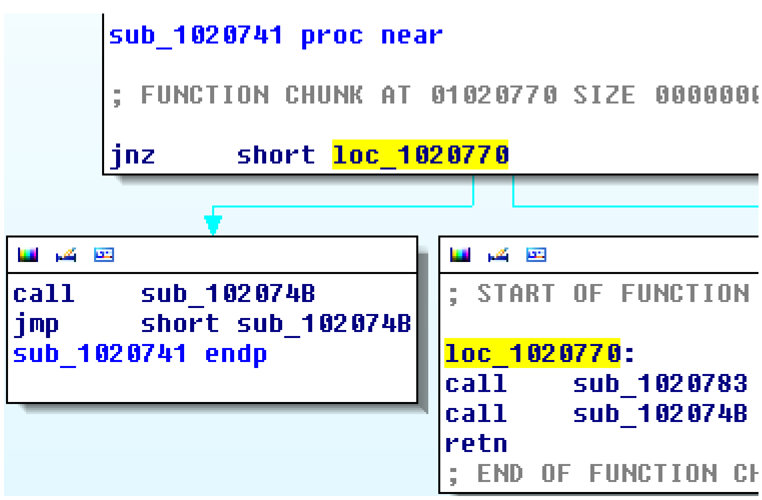
Removing the first layer

Dropping the program into IDA we immediately get a warning about the import segment being corrupted. We can ignore this since we already know that the program is packed.
We immediately begin seeing some fairly ugly code and the first few calls lead to the first trap.

```
sub_1020795 proc near
xor     eax, eax
push    dword ptr fs:[eax]
mov     fs:[eax], esp
dec     ebx
int     3               ; Trap to Debugger
retn
sub_1020795 endp ; sp-analysis failed
```

Because we entered this instruction through a call, and there is only one push, the next exception handler becomes the line after the last call at 0x01020714. This function and a similar one that uses 'inc ebx' instead, are called several times until a new call,

```
sub_1020736 proc near
cmp     ebx, 55h
call    sub_1020741
jmp     short sub_1020741
sub_1020736 endp
```



Earlier, 0x909055 was placed into ebx. Since only a few inc/dec traps were called, ebx != 0x55, so we will take the true (right) branch. Though it doesn't really matter since they both go to the same place.

```
sub_10207A8 proc near
xor     ebx, ebx
mov     ecx, 410C4Bh
sub     ecx, 40E301h
mov     edx, ebp
add     edx, 40E301h
lea     edi, [edx]
mov     esi, edi
xor     eax, eax
call    near ptr sub_10207CB
jmp     short near ptr sub_10207CB
sub_10207A8 endp
```

Here, an address is calculated and control is transferred there via a ret in the call.

After a bunch of decrypting code, we end up at 0x102084B, and control is transferred around using SEH, syscalls, interrupts, etc.

The resolve import loop happens at 0x01021477, which loads IMM32, ADVAPI32, RPCRT4, GDI32, and User32.

After some time, I discovered the reason that OllyDbg hangs when I try to run the program through to completion. The process suspends itself at 0x01021105. After struggling for a while, I made a pin tool to trace this section outside of a debugger:

<table>
<tr><td colspan="2">010210E2
010210E4
010210EA
010210ED
010210EE
010210F0
010210F2
010210F7
010210F9
010210FB
010210FD
010210FF</td></tr>
<tr><td>01021101
01021102
01021104
01021105
0102110A
0102110B</td><td>0102110D
0102110E
01021110
01021111
01021116</td></tr>
<tr><td colspan="2">01021117
01021118</td></tr>
</table>

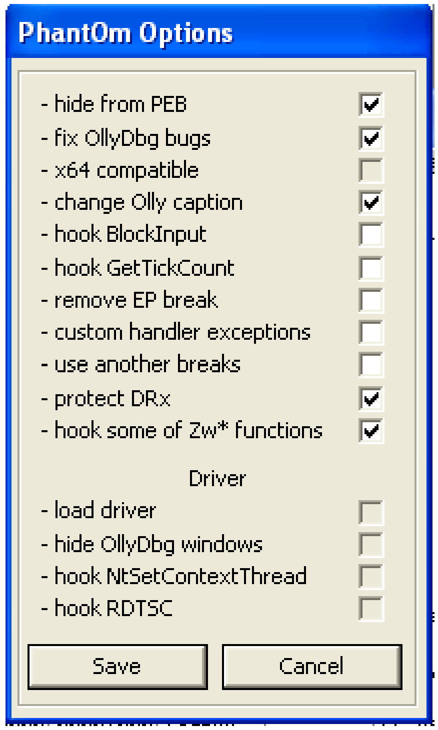
Figure 2: IP trace using Pin

It only happens twice, and it takes a different path each time. The first time through it pauses the thread, and the second time it resumes it. For whatever reason, only the left path is chosen in OllyDbg.

I wrote another pin tool to trace the path followed by the program outside a debugger in the section that eventually leads to the above section. For some reason, a critical decryption step is not taken when inside of a debugger.

I think a thread is supposed to spawn, do the decryption, and then resume the original thread.

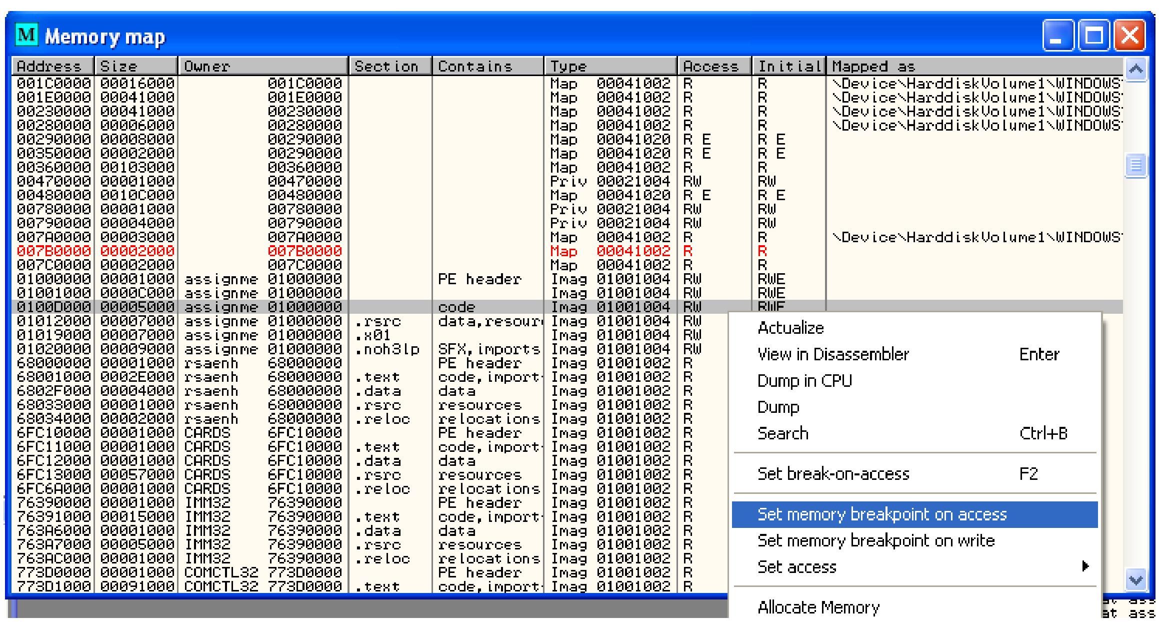
It turns out, that by using phant0m to hook the Zw* functions, we can bypass this failure.

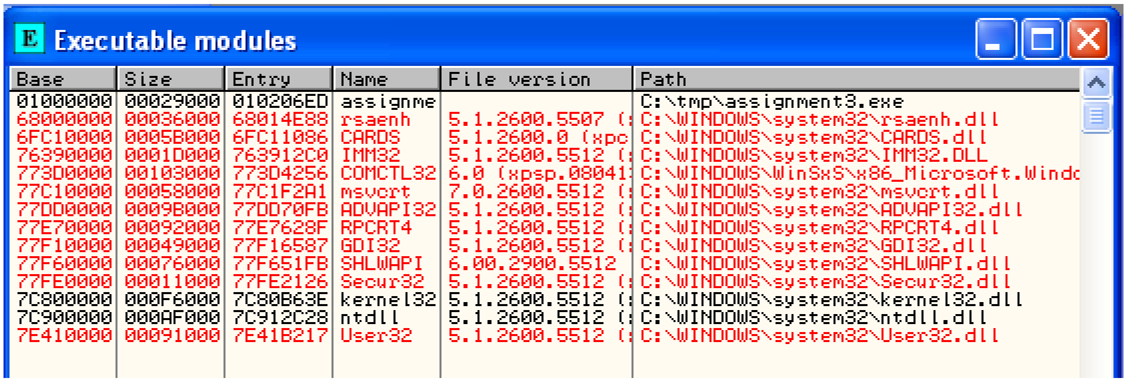


Now that we know where the failure is coming from, and how to mitigate, we can work on getting around the packer.

Lots of packing

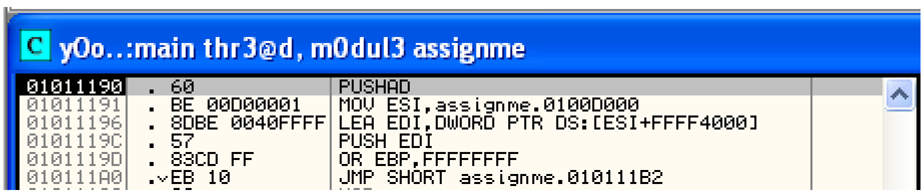
Since there is a nice section in mem map labeled code, let's set a section breakpoint with F2 and see what happens.



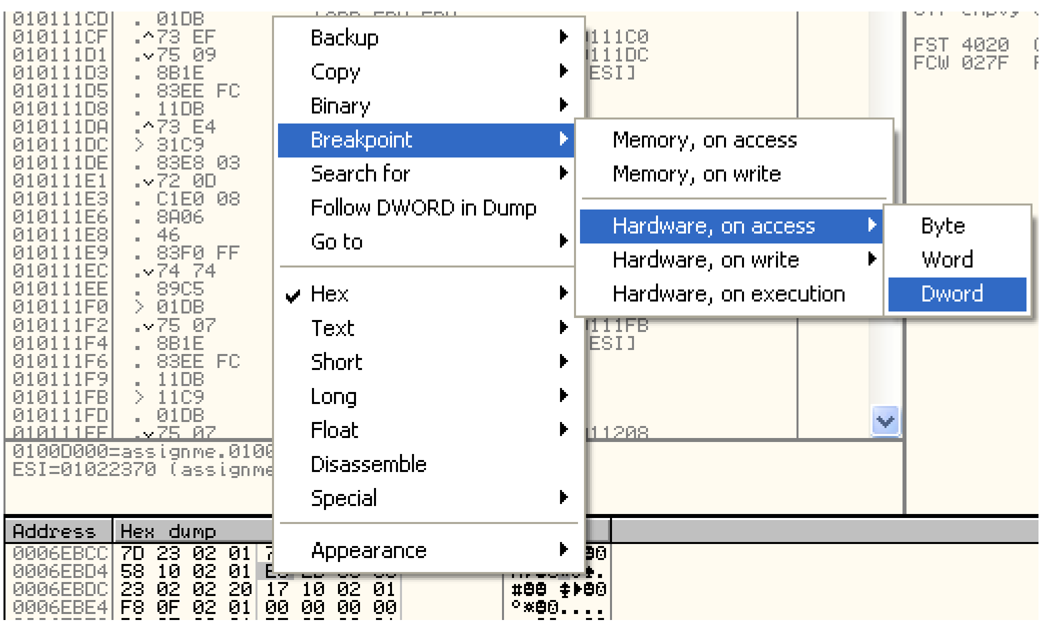


It looks like a few more modules were loaded, one being CARDS.dll. Since this is solitaire it is not a surprise. After hitting continue we get dropped at the beginning of another layer of packing.

The final layer



Awesome! A PUSHAD! These are nice because all we have to do is step, set a DWORD breakpoint on the stack, and continue to the corresponding POPAD. Right click ESP -> Follow in Dump, and then…



```
010113A9  . 61           POPAD
010113AA  . 8D4424 80    LEA EAX,DWORD PTR SS:[ESP-80]
010113AE  > 6A 00        PUSH 0
010113B0  . 39C4         CMP ESP,EAX
010113B2  .^75 FA        JNZ SHORT assignme.010113AE
010113B4  . 83EC 80      SUB ESP,-80
010113B7  .-E9 C94BFFFF  JMP assignme.01005F85
010113BC    00           DB 00
010113BD    00           DB 00
010113BE    00           DB 00
```

Boom, the corresponding POPAD with a far-call right afterwards, we were lucky!
Following the far-call leads us to the OEP, which is verified with VERA. Now that we know OEP, and we fixed the hanging from earlier, from now on we can just set a hardware breakpoint on 0x01005f85 and we'll be there!

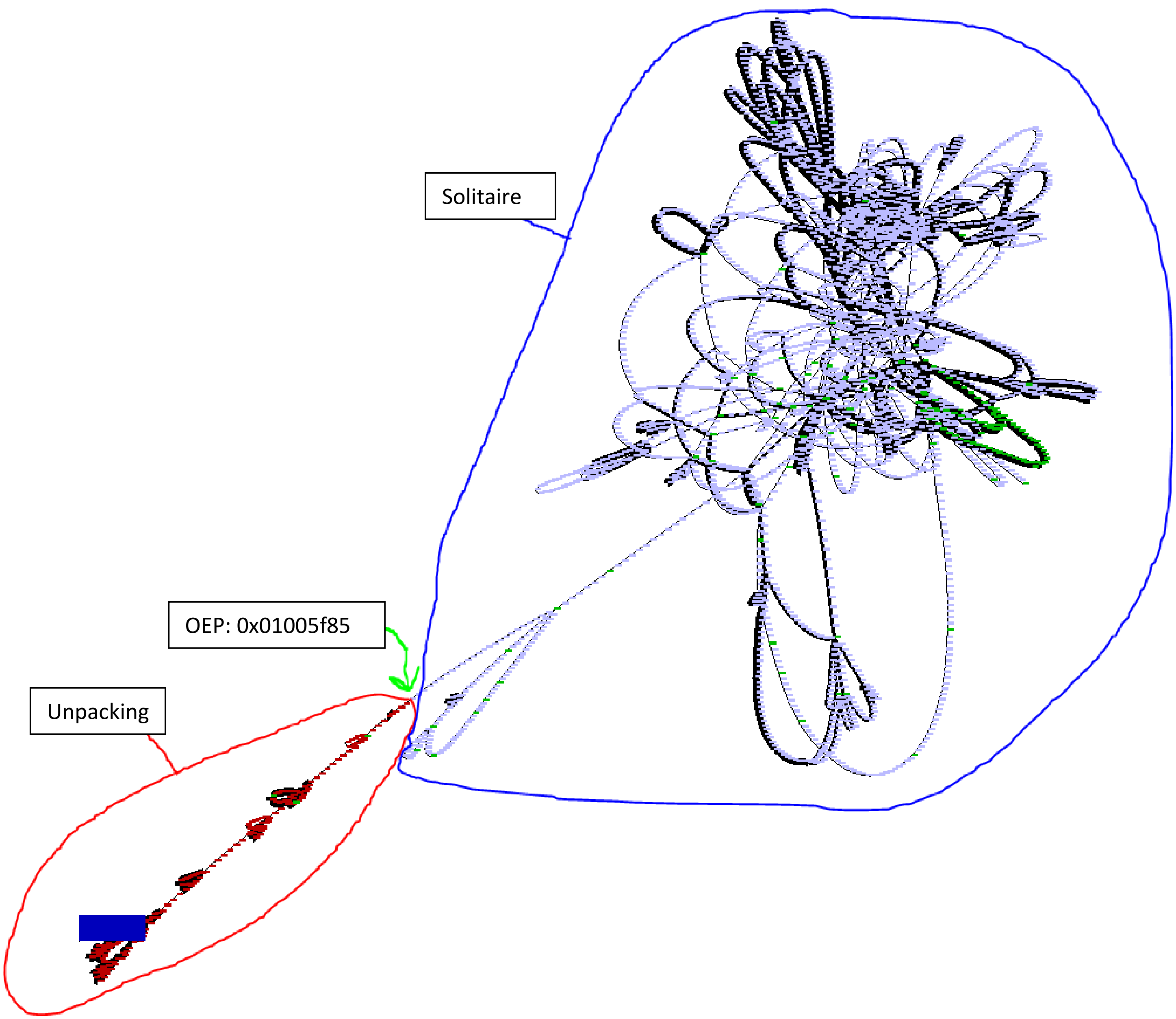
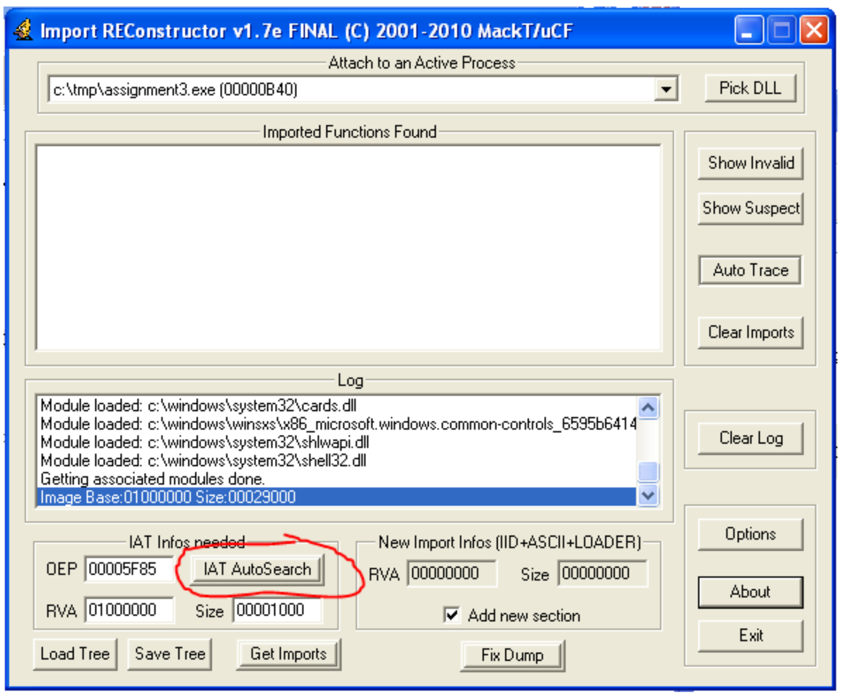


Figure 3: Made with VERA, a tool by Danny Quist.

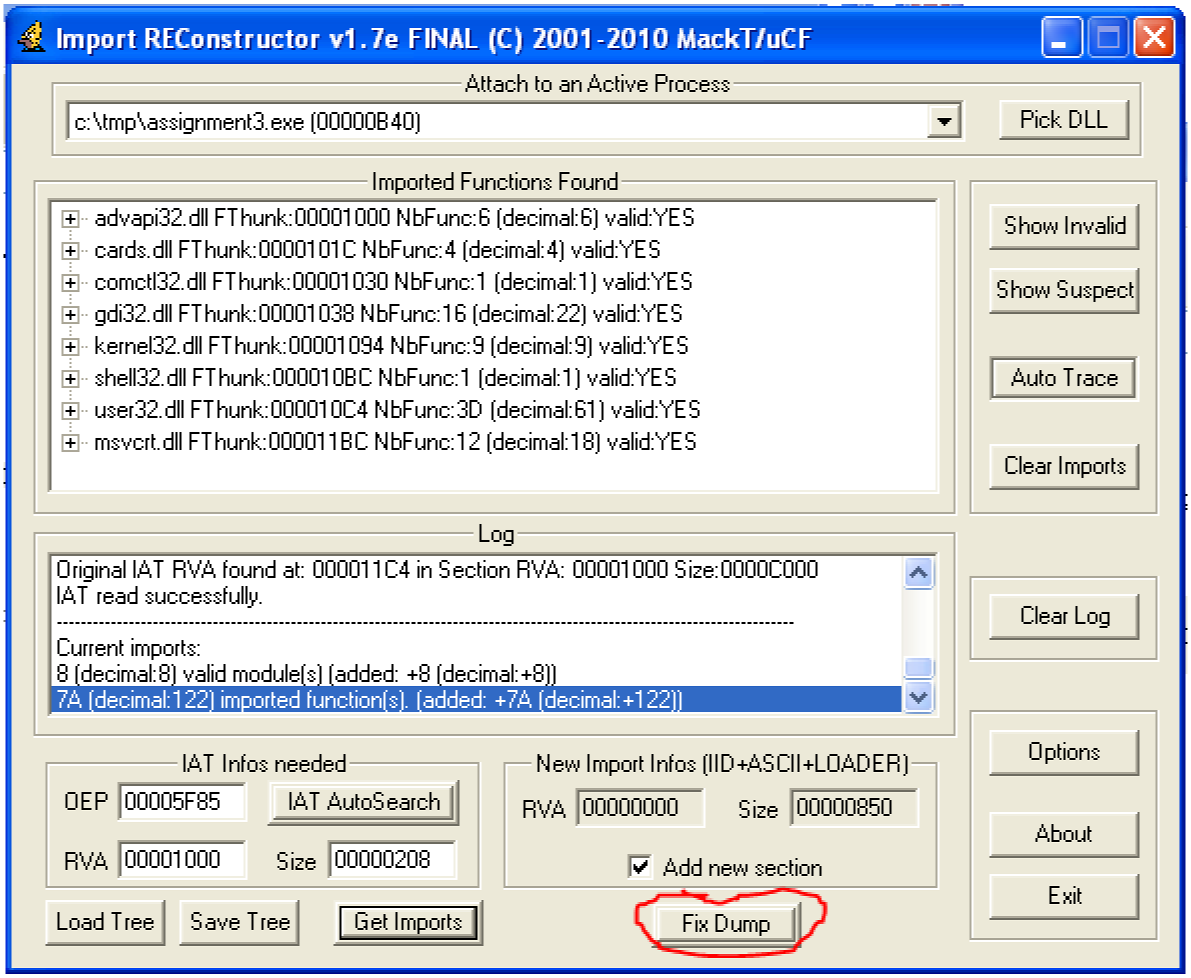
Dumping / Fixing the data

I used OllyDump as usual to dump the process from its OEP. There was a problem, though.

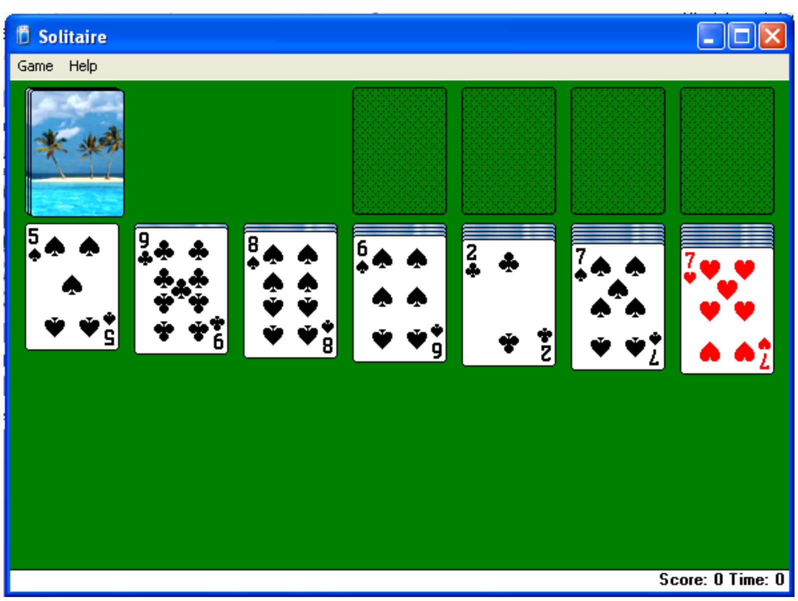
The import table isn't quite right. I decided to use a powerful tool called Import REConstructor (ImpREC).



Since I still have the program at OEP in OllyDbg, I selected it as the active process up top, changed the OEP and RVA boxes in the bottom left, and clicked 'IAT AutoSearch'.



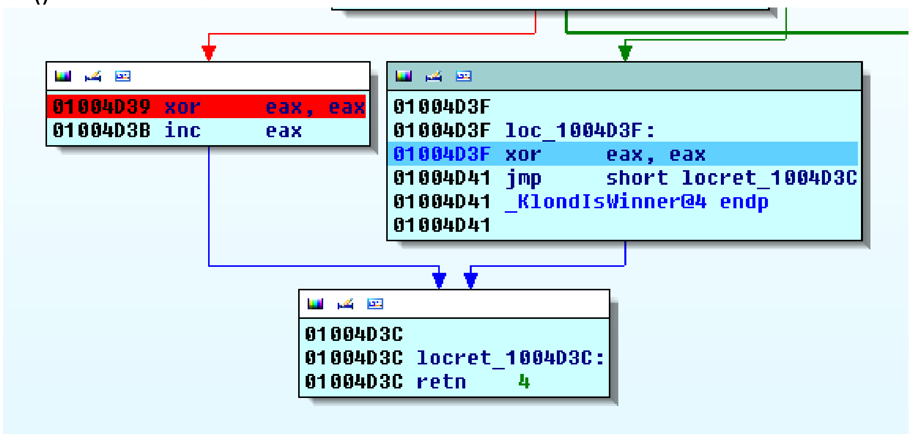
It automatically finds a bunch of valid imported functions. Now all that is left is to press 'Fix Dump' to fix our broken dump file. Success!



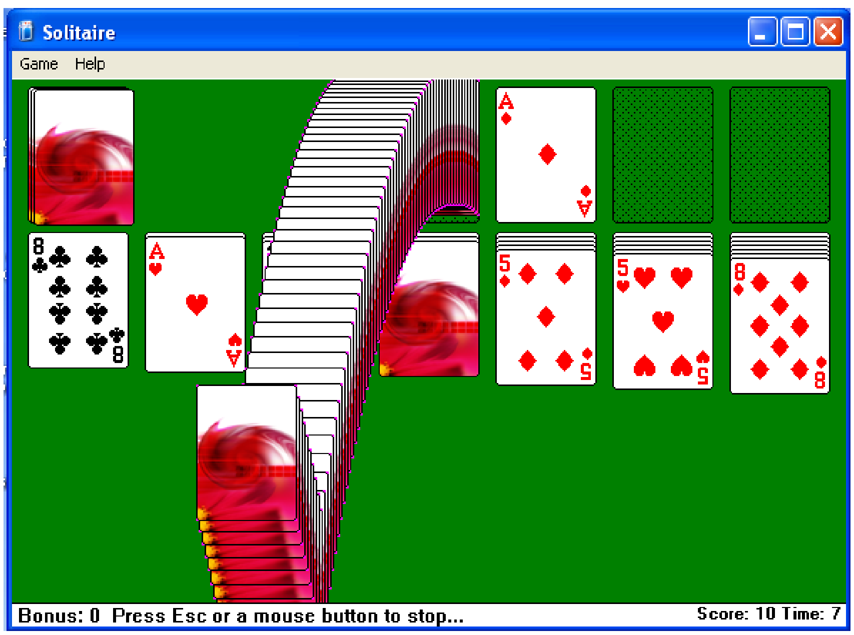
Patching for the win

Easy-win

Since the original solitaire has symbols, we can use that to aid in our process.
We find a KlondIsWinner() function at 0x1004D1E and at the bottom we see where the decision is made.



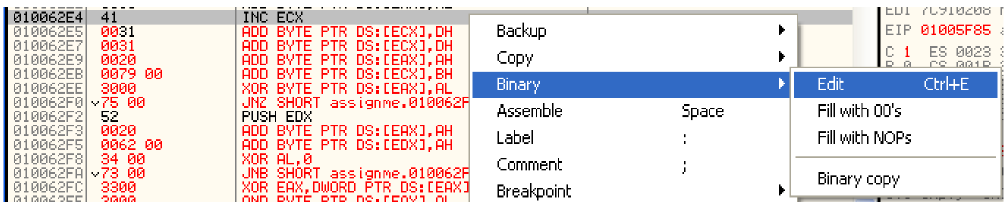
I set a breakpoint on each path, made a non-winning move, and was dropped in the right branch. It is easy to see now that this functions returns 1 when I am a winner, and 0 when I am not.
I chose to NOP both of the conditional branches above (not seen) so that the control flow always leads to the left branch.



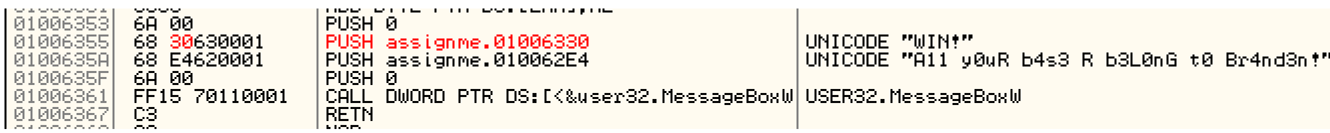
Credits

Clearly this awesome pwning can't go without some credit. A message-box sounds perfect.
We are going to utilize the code-cave technique to place our own message-box call inside of the win function that we patched earlier.
There are a bunch of NULL bytes starting at 0x10062D8, this is perfect for our code-cave. It also turns out that almost half of the executable is NULL bytes, meaning we can do a lot.

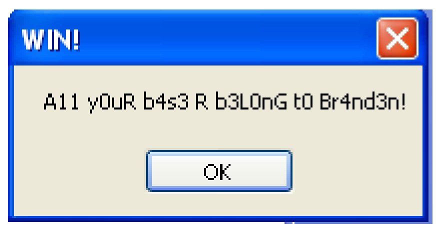
From MSDN, MessageBoxW() requires a window handle, body text, title text, and a type.



I used Binary -> Edit to add my body text and title text over the null bytes, making sure to use UNICODE since I am calling the wide version. The code inserted after is relatively simple, OllyDbg even picks up on our arguments.



Back inside of KlondIsWinner() I just added a 'call 01006353' so that this message-box appears before the famous dancing cards.



Automating the process

Unpacking

I wrote an OllyScript (find_oep.txt) to ask the user to confirm the important debug settings for OllyDbg and phant0m and go to the OEP. Once there the script informs the user that they can now dump the program from eip.
The imports are still broken and rather than have someone go through ImpREC I found an awesome tool called bsdiff. I used bsdiff to generate a patch file for the executable. I wrote patch.bat to take the dump from OllyDump, and use this patch file with bspatch to add the fix from ImpREC.

Patching

I used the same bsdiff/bspatch combo to patch the executable for the auto-win and add the message box from above. All that is needed after the program is dumped is to run
>.\patch.bat dump.exe
It will create several intermediate steps and produce a workable executable at the end.

Appendix

Pin tool (itrace.cpp)

Prints IP in a range. I used it to trace the program flow in certain sections outside of a debugger.

```
// Branden Clark
// Quick pin tool for tracing ip ranges

#include <stdio.h>
#include "..\..\include\pin\pin.h"
#include "..\..\include\pin\pin_isa.h"
FILE * trace;
void printip(void *ip) {
        if ((unsigned int)ip >= 0x10210E2 &&
                (unsigned int)ip <  0x1021120) {
                fprintf(trace, "%p\n", ip); }
        }

void Instruction(INS ins, void *v) {
   INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
}
void Fini(INT32 code, void *v) { fclose(trace); }
int main(int argc, char * argv[]) {
 trace = fopen("itrace.out", "w");
 PIN_Init(argc, argv);
 INS_AddInstrumentFunction(Instruction, 0);
 PIN_AddFiniFunction(Fini, 0);
 PIN_StartProgram();
 return 0;
}
```

OllyScript (find_oep.txt)

I used this with the OllyDbg plugin OllyScript to automate finding OEP.

```
// Branden Clark
// Find OEP in Assignment 3
// MD5:  989258474585D26352BAAC6FBFF91F0E assignment3.exe

var hwdBP // Local variable to store hardware breakpoint
var softBP // Local variable to strore software breakpoint

MSGYN "Is 'Hook some Zw* functions' selected in phant0m?"
cmp $RESULT,1
jne err
MSGYN "Is 'Hide from PEB' selected in phant0m?"
cmp $RESULT,1
jne err
MSGYN "Are all exceptions being passed to the program?"
cmp $RESULT,1
jne err
jmp start_proc:

start_proc:
mov hwdBP, 01005f85 // Store OEP to hardware breakpoint local variable
bphws hwdBP, "x" // Set hardware breakpoint (execute) on OEP
run // Run F9 command
cmt eip, "<<This is OEP>>"
msg "OEP found, you can dump the file starting from this address"
ret

err:
msg "Please check it accordingly"
ret
```

patch.bat

Automates the patching of IAT, win, and message-box

```
:: Branden Clark
:: Patch the dump from ollydbg/ollydump
@echo off

if "%1"=="" (
        echo Usage: "%~nx0 <dump_file.exe>"
        pause
        exit /b 1
)
for %%f in (%1) do (
        set orig=.\output\%%~nf
)
mkdir output
echo Copying dump file
copy "%1"  "%orig%"
echo Patching imports
.\bsdiff4.3-win32\bspatch  "%orig%" "%orig%_iat" ".\import_patch"
set orig_iat=%orig%_iat
echo Patching win
.\bsdiff4.3-win32\bspatch  "%orig_iat%" "%orig_iat%_win" ".\win_patch"
set orig_iat_win=%orig_iat%_win
echo Patching messagebox
.\bsdiff4.3-win32\bspatch  "%orig_iat_win%" "%orig_iat_win%_msgbox" ".\msgbox_patch"
set orig_iat_win_msgbox=%orig_iat_win%_msgbox
echo Rename result to an executable
copy "%orig_iat_win_msgbox%" ".\output\patched.exe"
```

ReadMe.txt

```
Instructions for automating the unpacking and patching process as much as
possible.

Unpacking
  o Get OllyDbg v1.10 and install the PhantOm, OllyDump, and OllyScript plugins.
  o Open assignment3.exe in OllyDbg
  o Options -> Debugging Options -> Exceptions
    - Pass all exceptions to the program
  o Plugins -> PhantOm
    - Select 'Hide from PEB' and 'Hook some Zw* functions'
  o Plugins -> OllyScript -> Run Script
    - select find_oep.txt
    - follow the prompts
  o Plugins -> OllyDump -> Dump debugged process
    - Double check OEP (0x1005f85)
    - Choose method 1 for rebuilding import table
    - Save as dump.exe

Patching
  o from cmd run
    - .\patch.bat dump.exe

The result is in .\output\patched.exe
```

Resources

Tools

Ollydbg v1.10: http://www.ollydbg.de/
VERA: http://www.offensivecomputing.net/?q=node/1687
ImportREC: https://tuts4you.com/download.php?view.415
MD5: 8899C9BC4E53B57726BE98ACF7936B62 ImpRec.exe
OllyScript: https://tuts4you.com/download.php?view.1418
MD5: 696B90A9EDE24B6950357C70D8155FAA OllyScript.dll
OllyDump: http://www.openrce.org/downloads/details/108/OllyDump
MD5: 8F30ED1E7BF42D6C70D16EA2E80E4448 OllyDump.dll
PhantOm: http://quequero.org/wp-content/uploads/2012/12/Phantom.zip
MD5: CCF7778A88B5F92519A456FC46BAA76B PhantOm.dll
bsdiff: http://sites.inka.de/tesla/download/bsdiff4.3-win32.zip
MD5: C3D26D0EA220CB1C73BAB57E2C526B77 bsdiff.exe
MD5: 2E7543A4DEEC9620C101771CA9B45D85 bspatch.exe

Docs

https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/index.html
http://www.cs.du.edu/~dconnors/courses/comp3361/notes/PinTutorial.pdf
http://x9090.blogspot.com/2009/07/ollyscript-tutorial-unpack-upx.html
http://www.scribd.com/doc/71638322/Weakness-of-the-Windows-API-part1