

Dumping & Analysis ActiveMark v6.2x

September 2007

1. Introduction

This document will focus on the changes relative to dumping and patching the new release of ActiveMark v6.2x. If you have not read my previous Tutorials on dumping and loading / patching AM v6.x Part I and II respectively, you are encouraged to do so prior to reading this. There is enough difference between the two versions to warrant another look. It seems the “Cat and Mouse” game continues in earnest. Many of the findings in my previous Tutorials have been remedied to be ineffective at best and useless at worst. The dumper and loader tools no longer work in the new release. New strategies have been employed to make dumping, analyzing and patching more difficult. Mind you, I am talking about a direct approach in dealing with a protection, not rebuilding a protected target application to be a virgin executable with the protection completely removed. I can only imagine what lies in wait for the new AM 2007 release.

This new version picks up where version 6.1x left off. This version employs some additional tricks such as using variables to store the PEB pointer of the originating executable on startup to discourage dumping. The use of GetTickCount API for discouraging active debugging and tracing. Saved GetTickCount values are stored in variables and used excessively between important functions. A low / hi sequence of instructions is called frequently to determine if we are operating through a debugger. AM surrounds functions using the GetTickCount API for refreshing variable pointers for determining the next course of operations. Whether this be a CALL DWORD PTR or a PUSH DWORD PTR with a RETN. As you might expect, by circumventing certain functional operations via tampering with code (i.e. NOP'ing JNZ's, JZ's or CALLS, etc.) you can quickly get into trouble as the DWORD variable that gets Called or Pushed is not correct. More troublesome is the continued CRC calculations and results stored in DWORD variables and used later on in processing to determine if the process is in sync (READ: DWORD variable such and such != some value) as well as to inhibit code modifications directly.

There appears to be more code extraction from the overlay layer and decryption of string literals to deter analysis. Some of the familiar processing remains, albeit with a new twist, while others are possibly missing or moved elsewhere in the processing curve. As is generally the case, many functions are obscured and difficult to trace. This is done by design to keep guarded code secret from prying eyes. AM also introduces a new technique to discourage using Hardware breakpoints. Some will simply not fire when set so it's quite clear they are being sabotaged. This appears to only be true in the 1st layer of processing. The code we are looking for may be buried in a long obfuscated routine, with PUSHAD's and POPAD's and winding GetTickCounts interspersed with JMP's, CALL's and PUSH / RETN's to make tracing more difficult.

People ask me, so how do you analyze this protection and why aren't you more detailed in your analysis? There is no simple answer. One starts out by opening a session of Notepad to keep painstaking notes as you debug and analyze. I can't tell you how many exceptions I encounter during the course of analysis. If something works, I jot it down for consideration and move on. It is difficult to explain something that is not tangible on the surface. The obfuscated code makes it difficult to point to a tangible structure and conceptualize the complete function or process flow. Finding the necessary conditions are hit and miss. The most important thing I can tell you is the Stack is your friend. You will need to analyze the Stack for the values and function returns that may be important. Tracing is also a valuable tool. How did I get here? After all is said and done, you compress your notes into something that makes sense enough to work on different target applications using the protection. Finally comes the writing of a Tutorial and putting all the pieces together that makes sense not only to you, who may have worked for many tedious hours analyzing the protection, but to the many people seeing this protection for the first time. The logical conclusion is that protection systems are meant to be difficult to understand and to analyze. To this end, AM achieves its goal.

I will be refraining from referencing an actual target application, Sorree!. This document is not intended to be a “crack” Tutorial. It is to provide general information that can be used to better understand the protection itself so that others can learn various methods effective in analyzing a protection system and further their knowledge base. With this in mind, let us proceed...

DUMPING & ANALYZING ACTIVEMARK V6.2X

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this document have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible for damages to the companies holding rights on those programs. The scope of this tutorial as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>

Table of Contents

Verification	2
1. Dumping ActiveMark v6.2x Applications, CondZero.....	3
1.1. Abstract	3
1.2. Target.....	3
1.3. How to dump.....	3
1.3.1 Preparation	3
1.3.2 Checking out the target.....	5
1.3.3 In the beginning.	5
1.4. Analysis	9
1.4.1 The Limitations	10
1.4.2 The Expiration	10
1.4.3 The Browser	15
1.4.4 The Final Condition.....	18
1.4.5 In The End.	18
1.4.6 Registry / Data key Build Function.....	19
1.5. Conclusions.....	20
1.6. Greetings	20

DUMPING & ANALYZING ACTIVEMARK V6.2X

1. Dumping ActiveMark v6.2x Applications, CondZero.

1.1. Abstract

This Tutorial will introduce you to the methods necessary in dumping and patching an ActiveMark v6.2x application so that you can further analyze the protection system and your application. At the time of this writing there are currently 2 releases: ActiveMARK[R] v6.2.478 and ActiveMARK[R] v6.2.505. The methods described in this document should be effective on both. Most distributed applications utilize 4 .text sections or layers eliminating the need for ImpRec and import reconstruction. Some have 3 layers which does not include the delayed imports option. For these you may require a tool such as ImpRec to rebuild imports. Once again we are interested in eliminating the need to process the 1st layer and use the 2nd layer EP as our entry / dumping point. We do however need to retain the original 1st layer EP as our entry point in the dumped application. Dumping and rebuilding imports has been covered in previous tutorials. I will not dwell too much on them here. The 3rd layer is divided into 2 sections. Pre and post processing. The Pre phase has to do with determining if the program is licensed or a Trial and the processing involved. The Post phase communicates directly with the main program through what AM refers to as “Code Blending”. This feature has stolen the real program’s OEP, stolen some imports, thus code will jump back and forth between the 3rd layer and the 4th layer (Main Program). There may also be some CRC checks included to prevent tampering. A few terms that we will be using to save keystrokes:

AM – ActiveMark

HWBP – Hardware Breakpoint (generally only 4 available)

SWBP – Software Breakpoint (F2 key in Olly on a line of code).

VM – Virtual Memory

TEB – Thread Environment Block

PEB – Process Environment Block

1.2. Target

N/A. There are various posts on the ARTEAM forum that discuss applications distributed with specific releases. Use the forum to find a suitable target.

1.3. How to dump

1.3.1 Preparation

Tools used: OllyDbg v1.10, Olly Advanced v1.26 Beta 10 for WinNT, Notepad. PETTOOLS v1.5 RC7 for dumping. XVI32 v2.51 Win hex editor for copying / pasting the overlay to the executable.

I will mention the options I use to run / analyze an AM target here as they may be important to you.

First, Ollydbg, options, Debugging options:

DUMPING & ANALYZING ACTIVEMARK V6.2X

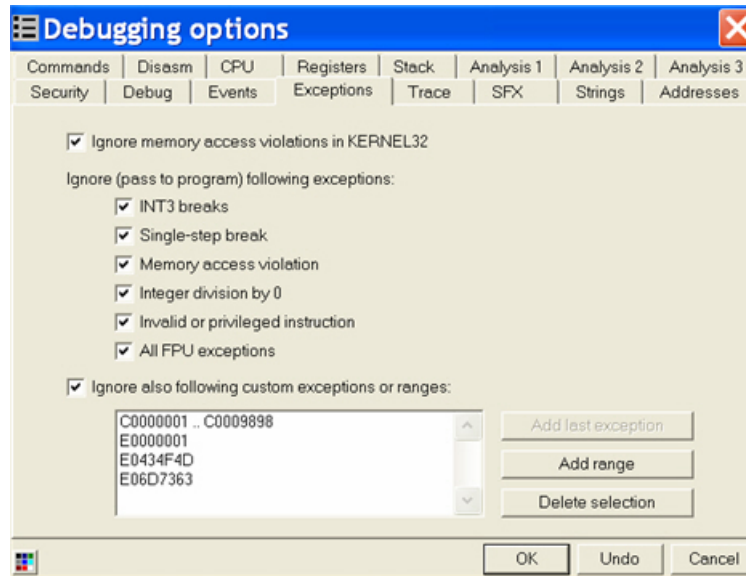


Figure 1.

and the Olly Advanced options Anti-Debug tab:

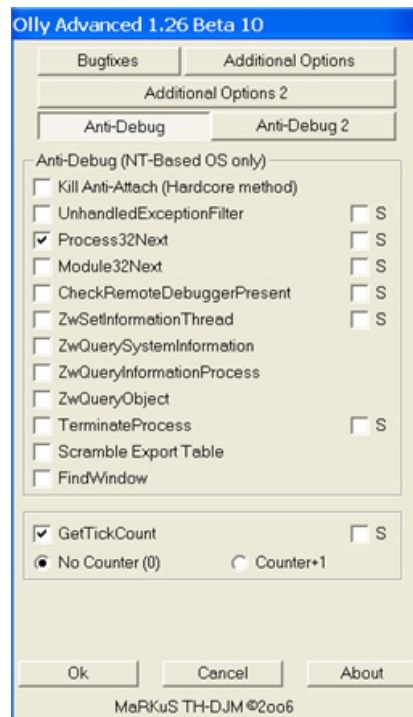


Figure 2.

I check off Process32Next and GetTickCount. The new version of AM utilizes both of these features quite Extensively. Process32Next (which is related to the CreateToolhelp32Snapshot API) is used to find software like Regmon, Filemon, LordPE and similar running as processes on your system. Also, it can detect if the target process was launched via Windows Explorer. GetTickCount surrounds virtually every major function. If one wants to single step through some code, then the represented machine times are going to reflect a long lapse in time which would indicate that a debugger may be attached to the process. We can circumvent that by selecting GetTickCount to increment time by no click. Of course we need to handle the IsDebuggerPresent API. We do so on the next tab (Anti-Debug 2) seen below:

DUMPING & ANALYZING ACTIVEMARK V6.2X

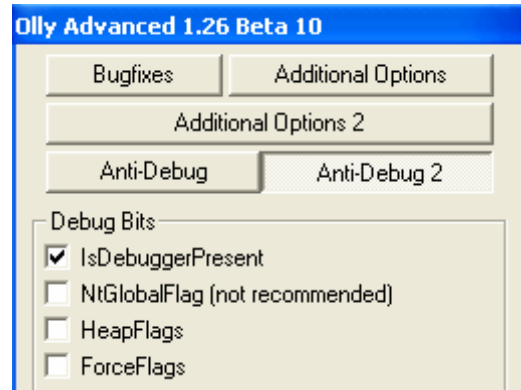


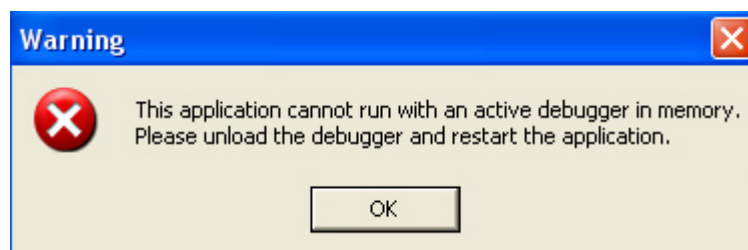
Figure 3.

1.3.2 Checking out the target

The selected target utilizes 4 layers of code (.text sections). Doing nothing else, if we run F9 the target in our debugger, Olly, we will get to the familiar “Browser Screen” (not shown). This indicates to us that our debugger is neither detected nor causing exceptions, but more importantly the Trial time left to evaluate an application, option to buy, language preference, etc. If you look at the stack in Olly, there is no clear indication how we arrived here. AM uses a lot of pushes to the stack with returns, as well as PUSHAD’s to make it difficult to determine the origin. Add the many API’s necessary to process java script and you have a vast number of dll’s being loaded to support that browser screen. All the associated AM project’s resources and methodologies are embedded and encrypted in the overlay. What I like to do is set a HWBP on the CreateFileA API. If nothing else, it provides a convenient starting point for observing what is happening as well as halts the process in the 3rd layer which we are most interested in after dumping. I covered how to arrive at the CALL for the browser screen in my previous Tutorial. Another consideration is to run the target normally, without a debugger, and set the option within the application to Windowed as opposed to “Full Screen”. This allows you to better monitor a debugged process in the post processing phase. The methods in this Tutorial fortunately are used prior to the actual application being launched (i.e. the 4th layer).

1.3.3 In the beginning...

The 1st layer is sneaky. This layer’s job is to thwart dumping and debugging an application, as well as decrypt / decompress the 2nd layer’s code section. To the untrained eye, seemingly very little is different. You still have the same verbiage in the beginning of the data section for that level. We still have the necessary use of retaining this section as our starting EP. A DWORD value is still saved in VM and which is integral to Layer 3 processing. This DWORD value is then stored in Virtual Memory via, you guessed it, a call to VirtualAlloc. You can generally find this value in the lower memory just above where the application is loaded. In my previous tutorial I mentioned to put a HWBP on the VirtualAlloc API and look for a size = 1000 hex bytes (4096 decimal). Well this has changed. AM nukes some of our HWBP’s so we are forced into using SWBP’s, but not on the 1st 2 bytes of the API. Try choosing the 5th or 6th byte and set your SWBP there. Also, we can no longer rely on a size = 1000, but instead look for everything with a size <= 1000. Follow the CALLS back into the source code where a certain DWORD value will be moved to this VM. AM saves the VM address generally to the last .bss section. Write down the address of code for future reference. For dumping purposes, we can reference an unused area in this same .bss section as the purported VM address to allow us to dump. For this we just move our desired .bss address to register EAX upon the return from VirtualAlloc. Before I forget, just beware that the use of SWBP’s on any code sections as well as the 1st 2 bytes of API’s can result in the following message:



DUMPING & ANALYZING ACTIVEMARK V6.2X

Figure 4

Once we get past the 1st layer your HWBP's will be effective. SWBP's will still be a problem from time to time because AM still CRC's code sections or parts of code sections to prevent tampering. SWBP = "CC" byte which is not the same as the original. A picture of the program's memory map will help guide us. Notice the number of sections. I will be referring to these names from time to time.

00400000	00001000			PE header	Imag	R	RWE
00401000	00019000	4th Layer	.text		Imag	R	RWE
0041A000	00007000		.rdata		Imag	R	RWE
00421000	00006000		.data		Imag	R	RWE
00427000	00018000		.rsrc		Imag	R	RWE
00442000	00245000	3rd Layer	.text	code	Imag	R	RWE
00687000	00008000		.data		Imag	R	RWE
0068F000	0004E000		.rdata		Imag	R	RWE
006DD000	0008F000		.bss		Imag	R	RWE
0076C000	00004000		.idata		Imag	R	RWE
00770000	0001C000		.rsrc		Imag	R	RWE
0078C000	0014F000	2nd Layer	.text		Imag	R	RWE
008DB000	00001000		.idata		Imag	R	RWE
008DC000	00019000		.rsrc	resources	Imag	R	RWE
008F5000	0002A000	1st Layer	.text	SFX	Imag	R	RWE
0091F000	0000D000		.bss		Imag	R	RWE
0092C000	00009000		.data	data	Imag	R	RWE
00935000	00001000		.idata	imports	Imag	R	RWE

Figure 5

Set a SWBP on the VirtualAlloc API by viewing the executables for kernel32 and viewing the names of this dll, selecting the VirtualAlloc API and following this entry in the disassembler like so:

7C809A81	8BFF	MOV EDI,EDI
7C809A83	55	PUSH EBP
7C809A84	8BEC	MOV EBP,ESP
7C809A86	FF75 14	PUSH DWORD PTR SS:[EBP+14]
7C809A89	FF75 10	PUSH DWORD PTR SS:[EBP+10]
7C809A8C	FF75 0C	PUSH DWORD PTR SS:[EBP+C]
7C809A8F	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7C809A92	6A FF	PUSH -1
7C809A94	E8 09000000	CALL kernel32.VirtualAllocEx
7C809A99	5D	POP EBP
7C809A9A	C2 1000	RETN 10

Figure 6

We can now run (F9) the target in Olly and I break here (I have purposely whited out the program name for reasons already mentioned:

Address	Value	Comment
0012FFA4	0091E0F9	.0091E0F9
0012FFA8	00900768	RETURN to .00900768 from .008FFE89
0012FFAC	00000000	Address
0012FFB0	00000312	Size
0012FFB4	00001000	AllocationType
0012FFB8	00000004	Protect
0012FFBC	0012FFE0	Pointer to next SEH record
0012FFC0	008F508E	SE handler
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F

Figure 7

Because of where the SWBP is set, the Parameters are not too clear, so I've included the necessary text for readability. Since the size is <= 1000 hex bytes we are interested in the Return. Follow the return in the disassembler to see if this is important or not. Set a SWBP on the code's return address. There can be more than 1 of these, but only the one associated with the DWORD value is important to us for dumping. Take note of register EAX because

DUMPING & ANALYZING ACTIVEMARK V6.2X

we are going to need to change this value to something within the program's VM, usually the last .bss section taken from the memory map we saw earlier when we dump. As it turns out this was not the one. The very next one WAS the one. You will code similar to the following:

009023A6	E8 6DF7FFFF	CALL .00901B18
009023AB	A3 B2C49200	MOV DWORD PTR DS:[92C4B2],EAX
009023B0	EB 02	JMP SHORT .009023B4
009023B2	80F8 EB	CMP AL,0EB
009023B5	01E8	ADD EAX,EBP
009023B7	C700 C26D1253	MOV DWORD PTR DS:[EAX],53126DC2
009023BD	BB 3E02AAC0	MOV EBX,C0AA023E
009023C2	EB 02	JMP SHORT .009023C6

Figure 8

There are 2 important things going on here. The red highlighted line is our return address where the VM address from Register EAX is moved to the 1st layer's .data section address 92C4B2. The 2nd item is the black highlighted line where a DWORD value C26D1253 is moved to the just allocated lower memory VM address. AM will refer back to the red item to obtain the VM address to get to the DWORD value in its 3rd layer processing. What we can do is simply change the value in Register EAX. If we were doing this "on the fly" via a dumper tool or Olly script, change the contents in address 92C4B2 to point to an address within the .bss section like so:

BEFORE: [0092C4B2]=003C0000

Olly's dump window for this VM address:

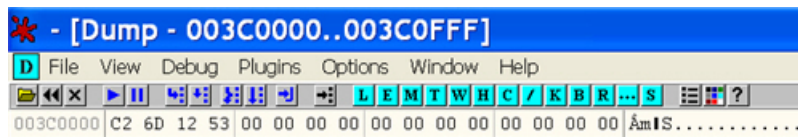


Figure 9

AFTER: [0092C4B2]=00923000

Address	Hex dump	ASCII
00923000	C2 6D 12 53 00 00 00 00	AmIS...
00923008	00 00 00 00 00 00 00 00	
00923010	00 00 00 00 00 00 00 00	
00923018	00 00 00 00 00 00 00 00	
00923020	00 00 00 00 00 00 00 00	

Figure 10

I have relocated the VM address "value" to our executable's 1st layer .bss section. Now when we dump, AM will always refer to this address. This logic was 1st introduced in AM v6.1x. When doing so, it is important to choose an area that will not be later overlaid with data. The 1st layer's .bss section has enough unused area for this purpose. If we were to build a dumper tool, we would need to assign increments of +1000 hex bytes to an original pointer to allow for all size <=1000 VirtualAlloc's, or follow the ASM code to determine which one contains the DWORD value move.

AM v6.2x continues the trend to protect the executable from being dumped by changing the section attributes via VirtualProtectEx. I switched to using PETOOL vs. LordPE as PETOOLS will allow me to dump the executable without a problem. I set a SWBP on this API and a Set memory BP on access for the 2nd layer's .text section to get to the 2nd layer EP. You may find another way more suitable. At this point, I think I'm done. To spare you the rest I will show you a new twist up front. Go into Olly's memory map window and view the upper limits of VM and the main thread's data block:

DUMPING & ANALYZING ACTIVEMARK V6.2X

7F6F0000	00007000				Map	R E	R E
7FFB0000	00024000				Map	R	R
7FFDE000	00001000	TEB		data block of main thread	Priv	RW	RW
7FFDF000	00001000	PEB			Priv	RW	RW
7FFE0000	00001000				Priv	R	R

Figure 11

And a dump of the TEB to show relevant data:

7FFDE000	0012FFBC	(Pointer to SEH chain)
7FFDE004	00130000	(Top of thread's stack)
7FFDE008	0012D000	(Bottom of thread's stack)
7FFDE00C	00000000	
7FFDE010	00001E00	
7FFDE014	00000000	
7FFDE018	7FFDE000	
7FFDE01C	00000000	
7FFDE020	00001584	
7FFDE024	00001374	(Thread ID)
7FFDE028	00000000	
7FFDE02C	00000000	(Pointer to Thread Local Storage)
7FFDE030	7FFDF000	Pointer to PEB - FS[30]
7FFDE034	00000006	(Last error = ERROR_INVALID_HANDLE)

Figure 12

Notice the address at: 7FFDE030 or TEB + hex 30 = FS[30]. AM v6.2x saves this pointer [7FFDF000] to the 3rd layer's .bss section normally 4 times during the 1st layer's processing (i.e. 4 different DWORD addresses). Below is a picture dump of 1 of these stored pointers in the 3rd layer's .bss section at address 006DD00C:

```
006DD000|00 00 00 00|00 00 00 00|00 00 00 00|00 F0 FD 7F|.....8yI
```

Figure 13

Each and every time you run the target, the TEB and PEB address may change, hence the PEB pointers will change. If you were to dump the application and run, your saved pointers may or may not work resulting in the dump crashing. You will need to find in memory these stored addresses in the 3rd layer's .bss section after we have reached our 2nd layer EP and where they are stored (using Notepad or similar), because we will need to address them in our dump. I noted 4, but this could change on the next release, so be aware of it.

LEVEL 2:

0078C9C9	8925 10C07800	MOV DWORD PTR DS:[78C010],ESP
0078C9CF	66:60	PUSHAW
0078C9D1	8925 1CC07800	MOV DWORD PTR DS:[78C01C],ESP
0078C9D7	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
0078C9DD	A3 18C07800	MOV DWORD PTR DS:[78C018],EAX
0078C9E2	E8 CBF8FFFF	CALL KimP.0078C2B2
0078C9E7	85C0	TEST EAX,EAX
0078C9E9	75 14	JNZ SHORT KimP.0078C9FF
0078C9EB	8B25 1CC07800	MOV ESP,DWORD PTR DS:[78C01C]
0078C9F1	A1 18C07800	MOV EAX,DWORD PTR DS:[78C018]
0078C9F6	64:A3 00000000	MOV DWORD PTR FS:[0],EAX
0078C9FC	66:61	POPAW
0078C9FE	C3	RETN

Figure 14

Okay, so we're at the 2nd layer's EP and ready to dump. You have saved the information for the DWORD value's VM address pointer and the 4 addresses used to store the PEB pointer in your Notepad, right? You dump your target (this had been covered previously) using your favourite PE tool (I used PETOOLS). After dumping and saving, you also would like to edit the PE file for the base of code and base of data. This is important for analysing and finding text references, not to mention following addresses easier in the disassembler's code window from the Stack.

DUMPING & ANALYZING ACTIVEMARK V6.2X

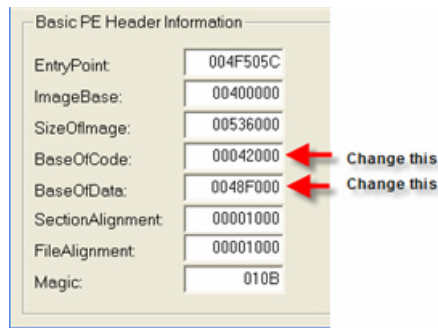


Figure 15

We do NOT change the Entry Point. The base of code reflects the 3rd layer's .text section. The base of data reflects the 3rd layer's .rdata sections respectively.

You did remember to copy and paste the overlay data to your dump, right? In some cases, the dumped.exe file may have to be renamed to the original. In this case, simply make a backup copy of the original before renaming your dump.

1.4. Analysis

There are some modifications we need to make to the dumped (unpacked) file in order for it to work properly. We saw that in AM v6.1x we needed to retain the original program's 1st layer EP and PUSH / RETN the 2nd layer's EP. We will additionally need to store the current processes' PEB pointers, plus the hi value used in the GetTickCount functions (I will get to this) before the PUSH / RETN as shown below:

008F505C	64:A1 30000000	MOV EAX,DWORD PTR FS:[30]	PEB pointer
008F5062	A3 0CD06D00	MOV DWORD PTR DS:[6DD00C],EAX	Move PEB pointer to saved 2nd layer .bss section
008F5067	A3 BF227400	MOV DWORD PTR DS:[7422BF],EAX	Move PEB pointer to saved 2nd layer .bss section
008F506C	A3 34287400	MOV DWORD PTR DS:[742834],EAX	Move PEB pointer to saved 2nd layer .bss section
008F5071	A3 D36E7600	MOV DWORD PTR DS:[766ED3],EAX	Move PEB pointer to saved 2nd layer .bss section
008F5076	C705 BCD27000 FFFFFFFF	MOV DWORD PTR DS:[70D2BC],-1	Move hi values to our hi saved GetTickCount variable
008F5080	68 C9C97800	PUSH .0078C9C9	Push 2nd layer's EP address
008F5085	C3	RETN	Go Now

Figure 16

The next order of business is the GetTickCount API. AM uses it a lot. (I referenced the important address above) More importantly, by using it they can determine if the process is being traced or debugged. While there is no way of knowing which 2 variables are being used at layer 2 EP dump time, it is only after the dump in the 3rd layer's processing that this becomes apparent. So if we were to NOP the line for address 008F5076 and run the dumped application in our debugger, it would terminate in the 3rd layer.

So we run our new dump with all the modifications in Figure 16 above, with the exception of GetTickCount in our debugger making sure that we have unchecked all exceptions and unchecked the GetTickCount option in Olly Advanced. We get to an exception:

Access violation when reading [00000000] - use Shift+F7/F8/F9 to pass exception to program

Figure 17

Doesn't make much sense. What went wrong? I decided to analyze the code section at this time. Normally, this isn't of much use because of the obfuscation. But in short doses, you can back trace a bit to find some potential problems. I got lucky and saw a GetTickCount API being pushed and then saw some DWORD pointers containing variables being compared to register EAX (the result from GetTickCount). I set a HWBP on the push to GetTickCount, reran the file and when I hit I followed the logic. Below is the main logic without the obfuscation for readability:

DUMPING & ANALYZING ACTIVEMARK V6.2X

```
CALL DWORD PTR DS:[6877D4] ; 0053857D
JMP 0053857D (POINTS TO CODE BELOW)
PUSHAD
PUSHFD
PUSH DWORD PTR DS:[72942C] ; kernel32.GetTickCount
CALL 004B3246
PUSH EAX
CMP EAX, DWORD PTR DS:[72026C]
JB 00538B4C
CMP EAX, DWORD PTR DS:[70D2BC]
JA 00538B4C
ADD ESP, 4
POPF
POPAD
```

There is usually a CALL to the function. The JB and JA above are “Bad Boy” jumps. We want to avoid them. In my case, the value in EAX was greater than the value in DWORD ptr [70D2BC]. I changed the value in this pointer to “FFFFFFF” when I reran and the process ran fine. This is why I included the MOV instruction in Figure 16 above. Note: You only need to do this if **NOT** using the Olly Advanced GetTickCount Count(0) option. You also wouldn’t need to do this if running the dumped application without a debugger attached. At this point now, we should have a fully working dumped / unpacked file if you make the changes noted in figure 16 above and save them to your dump.

You can refer to my previous Tutorial on dumping AM v6.x Part 1 for some more information on this subject. What we are doing here is expanding upon our base knowledge that has been accumulated since the advent of AM v6.x.

1.4.1 The Limitations

In analysing the target process using the methodologies obtained from AM v6.1x, it became clear that there were some differences. For the Expiration condition, it wasn’t clear from setting a HWBP on the text string “Implemented Categories” and then searching the stack for a Return that led to a TEST AL,AL condition, that there was one. Or if there was, AM made it so that any change to the value of AL “01” changed to “00” would lead to trouble. I say this because this is exactly the case for the Browser condition. Interestingly, some of these changes have an unintended benefit which will become clearer. The analysis and results that follow are just my conclusions. They are probably not perfect and there are better ways for sure. You can use them as a guide for your own analysis.

1.4.2 The Expiration

There is still a TEST AL,AL condition, however, it’s not the one or one’s that I would have guessed. I came upon 2 of them, more or less by accident and a lot of trial and error. The first and perhaps most useful in an ongoing dump / fix application returns the full time back to the Trial limitation “**FOR THE CURRENT EXECUTION ONLY!**”. But this could be considered okay, because that’s generally what you want to do each and every time you analyze the fixed target. The 2nd TEST AL,AL condition is generally reached after the Browser screen is deployed and then destroyed via “Start Trial” which I felt was unusual. Changing the value of AL from ‘01’ to ‘00’ will restore the full time and keep it, unlike the 1st. I will also show you a command you can search for which contains a pointer to the value currently stored for the results in processing the returned value from the GetSystemTimeAsFileTime API which AM uses for time comparisons. If you set this pointer value to all ‘0’s, you will get all your time back “For the current execution only”. So there we have it, 3 scenarios. Another interesting observation has to do with what I will call “Experimenting” with certain variables / registers and their values, conditions and of course, resetting the system clock and what can happen as a result and what you can do about it. There isn’t much change in the registry / data keys keeping track of the application’s usage. If you have a “NTFS” File system, AM also tracks this usage in the executable itself.

THE 1ST CONDITION – This Execution Only

You should see something similar to the following:

005F4CA6	84C0	TEST AL,AL
005F4CA8	59	POP ECX
005F4CA9	EB 01	JMP 005F4CAC

DUMPING & ANALYZING ACTIVEMARK V6.2X

AL=01 CHANGE TO '0'

This code is processed maybe 4 times in succession.

Looks familiar except for the POP ECX. What is interesting is that this condition is tested well in advance of most of the processing. You would hit this condition even before hitting the 1st CreateFileA BP and well before “Implemented Categories” and the registry / data keys processed. Finding this was a fluke, I guess, but you could easily do the same by testing virtually every TEST AL,AL condition to see the positive / negative benefits. Many of the TEST AL,AL conditions, as mentioned, are obscured by loops of code, push / returns, Calls to GetTickCount, etc. etc. Once you find them, easily done by the Search >> all commands >> command in Olly and setting conditional or non conditional SWBP's, it's just a matter of going through them all one by one as they are reached. The particular code I have indicated above can be found by doing the following:

SET a HWBP ON GetSystemTimeAsFileTime API. Take note of where this was called from. We are only interested in Calls from the program itself (i.e. not Calls from dll's).

Run F9 the program. Bypass the 1st call which appears highlighted below:

```

0046A747    FF15 E0C17600    CALL DWORD PTR DS:[76C1E0]
; kernel32.GetSystemTimeAsFileTime
0046A74D    8B75 FC          MOV ESI,DWORD PTR SS:[EBP-4]
0046A750    3375 F8          XOR ESI,DWORD PTR SS:[EBP-8]
0046A753    FF15 DCC17600    CALL DWORD PTR DS:[76C1DC]
; kernel32.GetCurrentProcessId
0046A759    33F0             XOR ESI,EAX
0046A75B    FF15 D8C17600    CALL DWORD PTR DS:[76C1D8]
; kernel32.GetCurrentThreadId
0046A761    33F0             XOR ESI,EAX
0046A763    FF15 D4C17600    CALL DWORD PTR DS:[76C1D4]
; kernel32.GetTickCount
0046A769    33F0             XOR ESI,EAX
0046A76B    8D45 F0          LEA EAX,DWORD PTR SS:[EBP-10]
0046A76E    50              PUSH EAX
0046A76F    FF15 D0C17600    CALL DWORD PTR DS:[76C1D0]
; kernel32.QueryPerformanceCounter

```

On the 2nd Call and BP you should see the following:

```

0044FEC1    55              PUSH EBP
0044FEC2    8BEC           MOV EBP,ESP
0044FEC4    51              PUSH ECX
0044FEC5    51              PUSH ECX
0044FEC6    56              PUSH ESI
0044FEC7    8D45 F8          LEA EAX,DWORD PTR SS:[EBP-8]
0044FECA    50              PUSH EAX
0044FECB    FF15 E0C17600    CALL DWORD PTR DS:[76C1E0]
; kernel32.GetSystemTimeAsFileTime
0044FED1    6A 01           PUSH 1
0044FED3    6A 00           PUSH 0
0044FED5    6A 00           PUSH 0
0044FED7    FF75 FC          PUSH DWORD PTR SS:[EBP-4]
0044FEDA    E8 31390100      CALL 00463810
0044FEDF    8B4D F8          MOV ECX,DWORD PTR SS:[EBP-8]
0044FEE2    33F6             XOR ESI,ESI
0044FEE4    03C1             ADD EAX,ECX
0044FEE6    13D6             ADC EDX,ESI
0044FEE8    A3 30C87300      MOV DWORD PTR DS:[73C830],EAX
0044FEED    8915 34C87300      MOV DWORD PTR DS:[73C834],EDX
0044FEF3    33C0             XOR EAX,EAX
0044FEF5    5E              POP ESI
0044FEF6    C9              LEAVE
0044FEF7    C3              RETN

```

DUMPING & ANALYZING ACTIVEMARK V6.2X

Execute the above function until the return. Do a Search, find all commands for TEST AL,AL, set log breakpoint on every command for condition AL==1 and F9 run the target. If you get the "Debugged program was unable to process exception" message (Due to encountering 'CC' byte during execution), scroll up to the offending instructions "84C0" (which represents the TEST AL,AL instruction, NOTE: the bytes may be included as part of another command) and write down this address and remember to remove it from the list of TEST AL,AL commands found and set a log BP on all commands found for condition AL==1 when you run the next time without this exception address. If rerunning for the exception, you may also want to set a HWBP on the RETN above and remove the HWBP on GetSystemTimeAsFileTime. You should hit the conditional TEST AL,AL on the 1st SWBP. If you found the right one, it will be executed maybe 4 times. The Browser screen will show you whether the full time has been restored or not. If not, then you found the wrong condition and must try again.

The 2nd Condition (Full Time)

After the BROWSER screen appears (Don't press any buttons, just yet!).

Do a Text Search for "Implemented Categories" and set a SWBP on all strings found.

Upon 1st BP look for the ASCII string that represents the registry / data key format in the stack shown below:

```
0012F5BC 0012F674
0012F5C0 00000026
0012F5C4 0000002F
0012F5C8 006A5760 ASCII "{%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X}"
0012F5CC 0106FEC0
0012F5D0 01135E98
0012F5D4 00000212
0012F5D8 0071A28C 0071A28C
```

Note your pushed address, mine was 006A5760 above. Remember, on your machine, the stack addresses as well as all address references in this tutorial, may be different.

Do a Search, all commands, for the command >> PUSH 006A5760 and set a SWBP on all commands found.

After the 1st BP is hit, remove all the SWBP's and do a new Search All commands for TEST AL,AL and

Set log breakpoint on every command for the following condition:

EAX==1

EBX==8

Which would look something similar to the following?

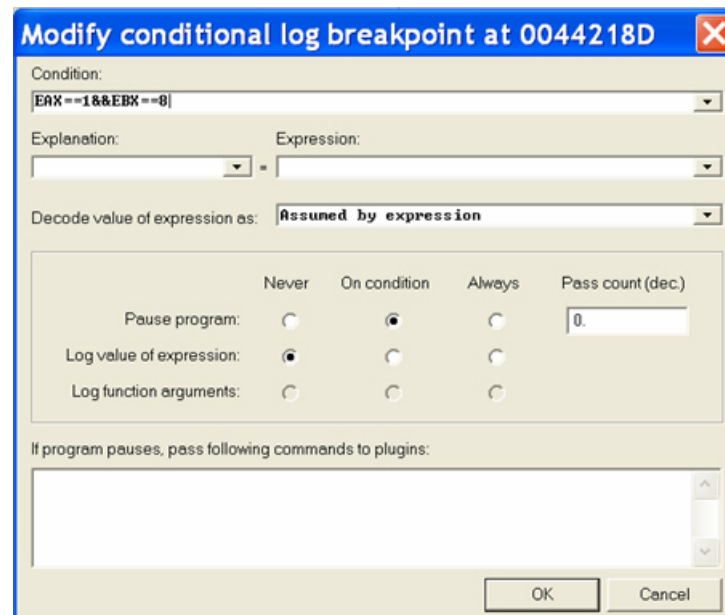


Figure 18.

DUMPING & ANALYZING ACTIVEMARK V6.2X

You will land on something similar to:

```
00593D90 > 84C0          TEST AL,AL
00593D92     EB 02        JMP 00593D96
AL=01 CHANGE TO '0'
```

Again, if you get into trouble and get the "Debugged program was unable to process exception" message (Due to encountering 'CC' byte during execution), either remove the offending address from your "found" list of commands or you can do the following:

Search down in the stack for the following and set a SWBP on the return address, mine is 0058337D shown below:

```
0012F6B0 |0058337D  RETURN to 0058337D from 005A7983
0012F6B4 |011CB800
0012F6B8 |0067EE97  0067EE97
0012F6BC |01259BB8
0012F6C0 |011D3AA4
0012F6C4 |0127B7CC
0012F6C8 |011D3AC0
0012F6CC |005D5647  RETURN to 005D5647
0012F6D0 |011CB804
0012F6D4 |012C54A0
0012F6D8 |01259BB8
0012F6DC |011D3AA4
0012F6E0 |011CB810
0012F6E4 |01267529  ASCII "MI"
0012F6E8 |01281034
0012F6EC |00577EB9  RETURN to 00577EB9 from 005968A8
0012F6F0 |007C503C  ASCII "PC"
0012F6F4 |01200580  ASCII "PC"
0012F6F8 |011CB930
0012F6FC |0012F728
0012F700 |00000000
0012F704 |3C000000
0012F708 |012675B0
0012F70C |012CB790
0012F710 |011D3AA4
0012F714 |01272CD0
0012F718 |0000000E
0012F71C |011CB9A8
0012F720 |011CB810
0012F724 |012C5608
0012F728 |011D3A7C
0012F72C |01279CBC
```

When you hit the BP on the return address, remove it, and do the Search >> all commands >> for the command TEST AL,AL and set a log BP on all commands for the condition EAX==1 && EBX==8 and rerun.

3rd Condition – VM DWORD value

This exercise is probably more useful if you would like to trace the resultant DWORD value which points to AM's result for the GetSystemTimeAsFileTime API.

Set a HWBP on the above API, and rerun the target. Note we want the 2nd BP hit for this API for the main program.

Do a Search >> all commands >> for the command: CMP DWORD PTR SS:[EBP-30],EBX
And set a SWBP on all commands found. Run F9 the target.

The 1st BP should look similar to the following:

```
00648E66  ^ 0F84 E1F3FFFF          JE 0064824D
```

DUMPING & ANALYZING ACTIVEMARK V6.2X

```

00648E6C    395D D0                CMP  DWORD PTR SS:[EBP-30], EBX
00648E6F    EB 03                JMP  00648E74
EBX=00000000

```

The value in the stack [EBP-30] is a pointer to a VM DWORD address.

Mine was Stack SS:[0012F510]=01166344

This address contained the following value:

```
01166344  29 B4 04 F7    Change to > 00 00 00 00
```

Note I changed the value to 0's and ran F9 the target. Full time was restored for the "current execution only".

BYPASS INTERNAL ERROR SCREEN

If for some unknown reason, this condition persists and won't go away, you can try the following. You are looking for the following condition:

```

005EDDD8    84C0                TEST AL, AL
005EDDDA    ^ 0F84 EDF5FFFF    JE  005ED3CD
005EDDE0    EB 05                JMP  005EDDE7
AL==01

```

AL should always == '01'. This value will be '00' for the internal error screen. Simply changing it to '01' will resolve the problem until the next time you do something wrong to trigger it.

To find it, set a HWBP on CreateFileA API.

Run F9 your target. You are specifically looking for the 1st BP where the stack looks like the following:

```

0012F3E8    00619120  /CALL to CreateFileA from 0061911E
0012F3EC    01135D88  |FileName = "C:\Program Files\Target
Directory\Target.exe\target.exe:{56DDDE5E-9C6C-462A-DAC5-C733D7F30679}"
0012F3F0    80000000  |Access = GENERIC_READ
0012F3F4    00000000  |ShareMode = 0
0012F3F8    00000000  |pSecurity = NULL
0012F3FC    00000003  |Mode = OPEN_EXISTING
0012F400    00000080  |Attributes = NORMAL
0012F404    00000000  \hTemplateFile = NULL

```

Here, AM is doing a read of the file to obtain a handle.

Look further down in the stack for the following: (Note: the format string will appear below ALL referenced registry / data key values that follow this format).

```

0012F5C8    006A5760  ASCII "{%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X}"
0012F5CC    0106FEC0
0012F5D0    01166300  UNICODE ".html:A]"
0012F5D4    00000212
0012F5D8    0071A28C  0071A28C
0012F5DC    01135C98
0012F5E0    00000000
0012F5E4    0000000F
0012F5E8    5EEDD56
0012F5EC    2A466C9C
0012F5F0    33C7C5DA
0012F5F4    7906F3D7
0012F5F8    0FF1520E
0012F5FC    21D7C740
0012F600    /0012F674

```

DUMPING & ANALYZING ACTIVEMARK V6.2X

0012F604 |005EDDD8 RETURN to 005EDDD8 from 0064A3A9

Follow the Return at address 005EDDD8 (Note: your machine / application may be different). Will lead to the TEST AL,AL condition. Alternately you can do a text search for "Implemented Categories" and set a SWBP on all strings found. Upon 1st BP look down in the stack for similar information shown above.

1.4.3 The Browser

AM has introduced a new twist on the TEST AL,AL condition for launching the browser. We would normally find the commands that precede this condition as follows:

Do a Search >> all referenced text strings >> "Implemented Categories" and set a HWBP on all strings found. After hitting the 1st BP, remove all HWBP's and do a Search >> all commands >> CALL DWORD PTR DS:[EAX+38] and set a SWBP on all commands found. Run the target. Usually the 2nd BP hit will look similar to what follows:

```
00623227    FF50 38                CALL DWORD PTR DS:[EAX+38]
DS:[006ABDF4]=0050D15C (0050D15C)
0062322A    52                    PUSH EDX
0062322B    68 582B6200          PUSH 00622B58
00623230    EB 02                JMP 00623234
```

You can step into the CALL above, or trace, to the following condition:

```
00622B67    84C0                TEST AL,AL
00622B69    EB 02                JMP SHORT 00622B6D
AFTER RESET AL == 0
```

Normally after changing the value of AL to '00', all is good. Not anymore. Change it and you are greeted with a Process Terminated message with little to go by in the stack. What we need to do is to keep our HWBP on the TEST AL,AL condition above and when we hit it, look down in the stack for the 2nd Return that precedes it.

TOP OF STACK:

```
0012F624 5B7638D8
0012F628 0105FEB0
0012F62C 0105FEB4
0012F630 01060058
0012F634 01163530 UNICODE
"tml?currenttime=2007/09/09@11:40:11&pooltime=2007/09/06@13:56:35&license_ts=0&contentid=eca6e7ac4b0
1"
0012F638 0012F850
0012F63C 7C90EE18 ntdll.7C90EE18
0012F640 7C9106F0 ntdll.7C9106F0
0012F644 FFFFFFFF
0012F648 7C9106EB RETURN to ntdll.7C9106EB from ntdll.7C90EE02
0012F64C 02838ED0 ASCII "http://amlocalhost.macrovision.com/offline/0071a5ac"
0012F650 003D0000
0012F654 40000060
0012F658 0000001C
0012F65C 00000033
0012F660 0000003F
0012F664 5B7638D8
0012F668 0012F850 Pointer to next SEH record
0012F66C 0067EE0F SE handler
0012F670 00000000
0012F674 /0012F68C
0012F678 |00484D34 RETURN to 00484D34
0012F67C |02838E98
0012F680 |02838EB0
0012F684 |00000000
```

DUMPING & ANALYZING ACTIVEMARK V6.2X

```
0012F688 |01068008
0012F68C |0012F6C8
0012F690 |00459BB1 RETURN to 00459BB1 from 00484BD6
0012F694 |0105FEB0
0012F698 |00622AE9 00622AE9
```

The 2nd Return at Address 00459BB1 above (yours will be different) needs to be followed in the disassembler window. You will trace into a large case structure the part we are most interested in is as follows:

```
00484D16 |. 0FBE4B 01      MOVZX ECX,BYTE PTR DS:[EBX+1]
00484D1A |. 83E9 44        SUB ECX,44
; Switch (cases 44..56)
00484D1D |. 5F            POP EDI
00484D1E |. 5E            POP ESI
00484D1F |. 5B            POP EBX
00484D20 |. 50            PUSH EAX
00484D21 |. 74 44         JE SHORT 00484D67
00484D23 |. 49            DEC ECX
00484D24 |. 49            DEC ECX
00484D25 |. 74 32         JE SHORT 00484D59
00484D27 |. 83E9 04       SUB ECX,4
00484D2A |. 74 19         JE SHORT 00484D45
00484D2C |. 83E9 0C       SUB ECX,0C
00484D2F |. 74 0F         JE SHORT 00484D40
00484D31 |. FF55 0C       CALL DWORD PTR SS:[EBP+C]
; Default case of switch 00484D1A
00484D34 |. 59            POP ECX
00484D35 |. 8B4D 14       MOV ECX,DWORD PTR SS:[EBP+14]
00484D38 |. 8345 14 04    ADD DWORD PTR SS:[EBP+14],4
00484D3C |. 8901          MOV DWORD PTR DS:[ECX],EAX
00484D3E |. EB 34         JMP SHORT 00484D74
00484D40 |> FF55 0C       CALL DWORD PTR SS:[EBP+C]
; 00499D12; Case 56 ('V') of switch 00484D1A
```

Set a HWBP on the last line above. Note: This is CASE 'V' of the switch.

Run the application without changing the preceding TEST AL,AL condition until the Browser screen appears. Press the Start button to continue with the Trial. Follow the CALL for the 2nd BP on the HWBP above. The code you are looking for should be similar to what follows although obfuscation may include some JMP statements in between:

THE FOLLOWING CONDITIONS MUST BE MET:

```
004C3120 A1 728A6800      MOV EAX,DWORD PTR DS:[688A72]
004C3125 EB 02        JMP SHORT 004C3129
```

NOTE: THE VALUE IN THE DWORD POINTER BELOW IS SIGNIFICANT!!

DS:[00688A72]=D67BB97C

EAX=028518B0

... Skip

```
004C313B 3305 5C766800    XOR EAX,DWORD PTR DS:[68765C]
004C3141 EB 02        JMP SHORT 004C3145
```

DS:[0068765C]=F36A406A

EAX=D67BB97C

... Skip

```
004C315A A3 728A6800      MOV DWORD PTR DS:[688A72],EAX
004C315F 3105 5C766800    XOR DWORD PTR DS:[68765C],EAX
004C3165 EB 02        JMP SHORT 004C3169
```

Note the value of EAX as **this** value must be exact

You will need to record the correct value [D67BB97C] in DWORD PTR: [688A72] and set a HWBP on the 1st encounter, address 004C3120 above.

DUMPING & ANALYZING ACTIVEMARK V6.2X

I call this code a checkpoint. AM is doing this with more frequency. If certain code is not processed within a loop or sequence of functions are bypassed, then the variable is not updated (retains some old value) and you are bounced out. It is important that we set this variable with the correct data or we can't continue. Even so, after fixing this small problem, we still have another problem to deal with. This is what I'll call the...

"Good Boy / Bad Boy" condition

```
004C2DB1  > \84C0          TEST AL, AL
004C2DB3  . EB 03          JMP SHORT 004C2DB8
AL=00
BL=01
ABOVE VALUES MUST BE AS SHOWN. IF REGISTER EBX IS '0'
YOU HAVE A PROBLEM.
```

Yep, another TEST AL,AL, only this time AL must = '00' and BL must = '01'.

There are a couple ways to find the code above.

1st method:

After reaching the HWBP in the checkpoint above,

Do a Search >> all commands for either of the statements below and set a SWBP on all commands found:

LEA EAX, DWORD PTR SS:[EBP+48]

OR

LEA ECX, DWORD PTR SS:[EBP+48]

When you hit the SWBP above, you can set a conditional trace on TEST AL,AL where AL==0&&BL==0

Assuming that you changed to original Browser conditional TEST AL,AL to '0'. If not, then the conditional values would be AL==0&&BL==1.

An alternate approach is to look for the following procedure which is generally not obfuscated:

```
0044BA64  /$ 56          PUSH ESI
0044BA65  |. 8BF1         MOV ESI, ECX
0044BA67  |. 8B06         MOV EAX, DWORD PTR DS:[ESI]
0044BA69  |. 85C0         TEST EAX, EAX
0044BA6B  |. 57          PUSH EDI
0044BA6C  |. 8B7C24 0C    MOV EDI, DWORD PTR SS:[ESP+C]
0044BA70  |. 74 04        JE SHORT 0044BA76
0044BA72  |. 3B07        CMP EAX, DWORD PTR DS:[EDI]
0044BA74  |. 74 05        JE SHORT 0044BA7B
0044BA76  |> E8 EDC3FFFF  CALL 00447E68
0044BA7B  |> 8B46 04      MOV EAX, DWORD PTR DS:[ESI+4]
0044BA7E  |. 2B47 04      SUB EAX, DWORD PTR DS:[EDI+4]
0044BA81  |. 5F          POP EDI
0044BA82  |. F7D8        NEG EAX
0044BA84  |. 1BC0        SBB EAX, EAX
0044BA86  |. 40          INC EAX
0044BA87  |. 5E          POP ESI
0044BA88  \. C2 0400     RETN 4
```

Set a conditional SWBP on the RETN 4 above address 0044BA88.

For the condition EAX==0&&EBX==1

Step into the 3rd Return.

Or, you can do a Search >> all commands >> TEST AL,AL with the log BP for condition EAX==0&&EBX==1.

Look for the 4th BP found.

DUMPING & ANALYZING ACTIVEMARK V6.2X

You will need to ensure that when the “Good Boy / Bad Boy” TEST AL,AL condition at address 004C2DB1 is reached via a HWBP, that you move a value of ‘01’ to BL to continue. These are the steps necessary to eliminate the Browser screen.

There is one unintended side-effect in the above steps. We no longer need to worry about the TIMER / TIMEOUT conditions. They are eliminated because of the code we bypassed in dealing with the Browser screen.

1.4.4 The Final Condition

I bring this to your attention as an FYI kind of thing. You shouldn’t normally need to worry about it. After you reach the HWBP checkpoint condition above, you can do a Search >> all commands >> TEST AL,AL for the conditions: EAX==1&&EBX==1 and set a SWBP on all commands found.

Normally on the 7th BP reached you will see the following:

```
0060928F      84C0                      TEST AL,AL
00609291      EB 03                      JMP SHORT 00609296
The following values should be found:
AL=01
BL=01
```

Normally on the 6th BP you will notice you are getting close as Register EDX will contain the following ASCII string:

EDX 0012F878 ASCII "--LocalServer"

And the next TEST AL,AL will be the one.

1.4.5 In The End...

If we followed the code after reaching the HWBP above for the Final Condition, we eventually will reach this code:

```
005FED54      FFE0                      JMP EAX
```

This is where we jump to our post processing phase in the 3rd layer which leads to the main program. If all has been done correctly, then the value of EAX is within the 3rd layer’s code section, otherwise, you can guess the rest. The code snippets that follow are a prelude to the 3rd layer’s post processing:

PRELUDE TO MAIN PROCESSING:

```
0060310F      A1 C83A7300                MOV EAX,DWORD PTR DS:[733AC8]
00603114      64:A3 00000000            MOV DWORD PTR FS:[0],EAX
0060311A      61                        POPAD
0060311B      EB 03                      JMP SHORT 00603120

00526F9B      6A 60                      PUSH 60
00526F9D      68 58BE4100                PUSH 0041BE58

00407706      E8 BD100000                CALL 004087C8
0040770B      BF 94000000                MOV EDI,94
00407710      8BC7                      MOV EAX,EDI
00407712      E8 49F5FFFF                CALL 00406C60
00407717      8965 E8                    MOV DWORD PTR SS:[EBP-18],ESP
0040771A      8BF4                      MOV ESI,ESP
0040771C      893E                      MOV DWORD PTR DS:[ESI],EDI
0040771E      56                        PUSH ESI
0040771F      FF15 E8C27600                CALL DWORD PTR DS:[76C2E8]
; kernel32.GetVersionExA
```

DUMPING & ANALYZING ACTIVEMARK V6.2X

1.4.6 Registry / Data key Build Function

Some of you may remember the following code from a previous version 5.x Advanced Tutorial.

00484B3A	55	PUSH EBP
00484B3B	8BEC	MOV EBP, ESP
00484B3D	81EC 04010000	SUB ESP, 104
00484B43	A1 A4766800	MOV EAX, DWORD PTR DS:[6876A4]
00484B48	33C5	XOR EAX, EBP
00484B4A	8945 FC	MOV DWORD PTR SS:[EBP-4], EAX
00484B4D	56	PUSH ESI
00484B4E	8BF1	MOV ESI, ECX
00484B50	0FB646 0F	MOVZX EAX, BYTE PTR DS:[ESI+F]
00484B54	50	PUSH EAX
00484B55	0FB646 0E	MOVZX EAX, BYTE PTR DS:[ESI+E]
00484B59	50	PUSH EAX
00484B5A	0FB646 0D	MOVZX EAX, BYTE PTR DS:[ESI+D]
00484B5E	50	PUSH EAX
00484B5F	0FB646 0C	MOVZX EAX, BYTE PTR DS:[ESI+C]
00484B63	50	PUSH EAX
00484B64	0FB646 0B	MOVZX EAX, BYTE PTR DS:[ESI+B]
00484B68	50	PUSH EAX
00484B69	0FB646 0A	MOVZX EAX, BYTE PTR DS:[ESI+A]
00484B6D	50	PUSH EAX
00484B6E	0FB646 09	MOVZX EAX, BYTE PTR DS:[ESI+9]
00484B72	50	PUSH EAX
00484B73	0FB646 08	MOVZX EAX, BYTE PTR DS:[ESI+8]
00484B77	50	PUSH EAX
00484B78	0FB646 07	MOVZX EAX, BYTE PTR DS:[ESI+7]
00484B7C	50	PUSH EAX
00484B7D	0FB646 06	MOVZX EAX, BYTE PTR DS:[ESI+6]
00484B81	50	PUSH EAX
00484B82	0FB646 05	MOVZX EAX, BYTE PTR DS:[ESI+5]
00484B86	50	PUSH EAX
00484B87	0FB646 04	MOVZX EAX, BYTE PTR DS:[ESI+4]
00484B8B	50	PUSH EAX
00484B8C	0FB646 03	MOVZX EAX, BYTE PTR DS:[ESI+3]
00484B90	50	PUSH EAX
00484B91	0FB646 02	MOVZX EAX, BYTE PTR DS:[ESI+2]
00484B95	50	PUSH EAX
00484B96	0FB646 01	MOVZX EAX, BYTE PTR DS:[ESI+1]
00484B9A	50	PUSH EAX
00484B9B	0FB606	MOVZX EAX, BYTE PTR DS:[ESI]
00484B9E	50	PUSH EAX
00484B9F	FF76 2C	PUSH DWORD PTR DS:[ESI+2C]
00484BA2	8D85 FCFEFFFF	LEA EAX, DWORD PTR SS:[EBP-104]
00484BA8	68 FF000000	PUSH 0FF
00484BAD	50	PUSH EAX
00484BAE	E8 6D00FFFF	CALL 00474C20
00484BB3	83C4 4C	ADD ESP, 4C
00484BB6	8D85 FCFEFFFF	LEA EAX, DWORD PTR SS:[EBP-104]
00484BBC	50	PUSH EAX
00484BBD	8D4E 10	LEA ECX, DWORD PTR DS:[ESI+10]
00484BC0	C645 FB 00	MOV BYTE PTR SS:[EBP-5], 0
00484BC4	E8 F3FD0800	CALL 005149BC
00484BC9	8B4D FC	MOV ECX, DWORD PTR SS:[EBP-4]
00484BCC	33CD	XOR ECX, EBP
00484BCE	5E	POP ESI
00484BCF	E8 5860FCFF	CALL 0044AC2C

DUMPING & ANALYZING ACTIVEMARK V6.2X

00484BD4	C9	LEAVE
00484BD5	C3	RETN

It's still there although a little bit larger now. If you put a RETN at the top of this function (address 00484B3A) shown above, the registry / data keys are not generated. You do need to remove the code after doing this and reapply again every time this function is called, so I'm not sure how many will want to use it.

1.5. Conclusions

There are some new features in ActiveMark v6.2x to learn and control. This Tutorial is meant as a learning tool to demonstrate the most basic method to dump an AM v6.2x application and analyze the protection. If, after reading this Tutorial, better methods in dumping AM v6.2x applications can be realized, then the author has accomplished his goal. There are obvious extensions to this document. How best to inline code any changes? The creation of an automated dumping tool or script perhaps. More questions may be raised than answers given. I hope you, the reader, enjoyed this learning experience and hopefully one day write a Tutorial of your own.

1.6. Greetings

I wish to thank all the ARTeam members and of course you who, have read this tutorial and perhaps, can contribute something worthwhile to the RCE community.



<http://forums.accessroot.com/>