

Dumping ActiveMark v6.x – PART I

December 2006

1. Introduction

Many of you are by now familiar with ActiveMark now incorporated as part of Macrovision® Corporation. This software protection system improves upon previous versions by employing more strategic code obfuscation and literal string encryption. This new version now allocates memory to include a DWORD value used in conjunction with the CPUID instruction to enforce running the executable on a single machine to thwart the distribution of any pirated / copied applications (if so enabled). ActiveMark has also beefed up its security protection system by utilizing active memory scanning and specific AM code section check summing to inhibit modification of its code. This may be the single most important and significant new change to date in its arsenal. This is obviously meant to discourage modifications to its code in unpacked / dumped applications as occurred in the past. What this all means to you is the dawn of a new era of ActiveMark sophistication in its software protection system that requires new methods in analyzing and understanding the implications and practical use thereof.

Another interesting feature, in this new release, is what can be best described as the separation of the previous generation's 2nd layer code into, what is now, two distinct new layers. Where as in the past, the 1st layer would decompress / decrypt ALL the code for an application, in the new version, the 1st layer decompresses / decrypts the code for the new 2nd layer, which in turn decompresses / decrypts / protects the code for the 3rd layer and the original code, albeit with numerous imbedded calls to the AM 3rd layer as in the past. So now, a new 3rd layer communicates with the original code. Of course this further bloats any executable with resultant performance degradation. Also, the new 3rd layer protection changes the characteristics of each section in the PE executable to inhibit the dumping of any application using a 3rd party dumping tool (i.e. LordPE)

This is Part I in introducing measures to effectively dump and analyze the new AM release v6.x. In Part II, we will discuss the necessary steps to patch and run a target process under this protection.

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.x 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
1.3.4 Analyzing the target	11
1.3.5 The AMDUMPV6 (Dumper Tool)	11
1.4. References	12
1.5. Conclusions	12
1.6. Greetings	12

1. Dumping ActiveMark v6.x Applications, CondZero.

1.1. Abstract

This Tutorial will introduce you to the methods necessary in dumping an ActiveMark v6.,x application so that you can further analyze the protection system and your application. Included is a new tool “AMDUMPV6” which can aid you in dumping an application and show you some important concepts in dumping an application.

1.2. Target

The target is a game called Buku kakuro - 31.3 Mb - 6.1.342 AM release. You can get it at the link below:
http://d.trymedia.com/dd/merscom/60m_d/merscom/BukuKakuroSetup.exe

1.3. How to dump

1.3.1 Preparation

Tools used: OllyDbg v1.10, OllyAdvanced v1.26 Beta 10 for winnt, AMDUMPV6.

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:

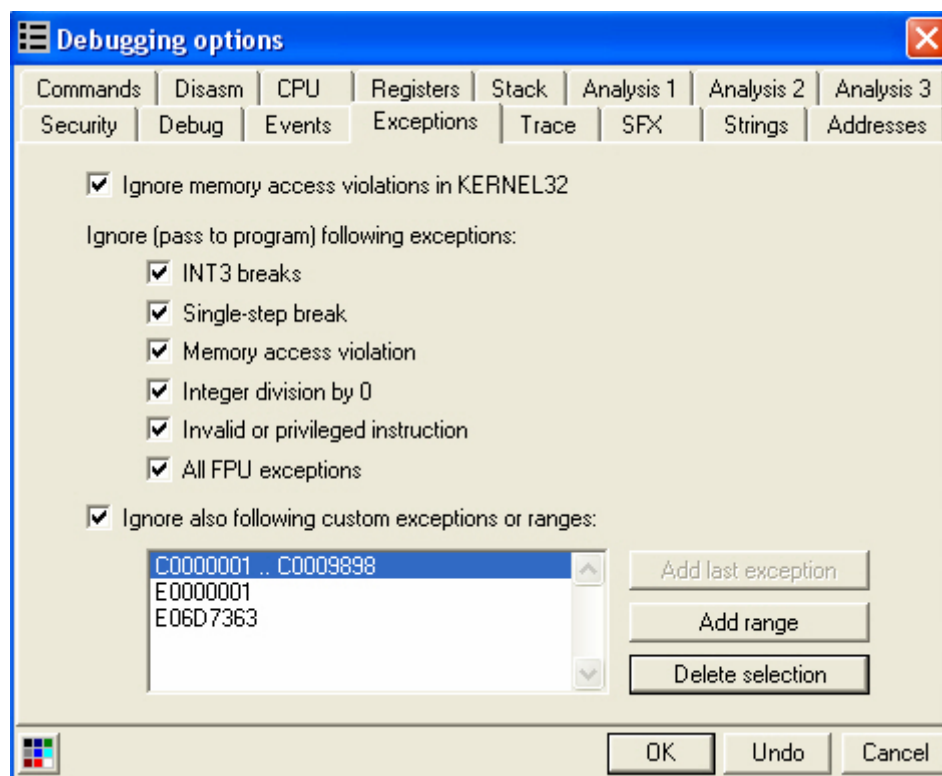


Figure 1.

I like to check everything. Notice (2) entries in the custom exceptions box:

1. E0000001
2. E06D7363

For some apparent reason, AM likes to use these 2 exceptions to denote a stage in the process. They Handle these exceptions and I choose to ignore them.

PRIMER ON DUMPING ACTIVEMARK V6.X

Second, using the Olly Advanced plugin:
First, the Anti-Debug options:

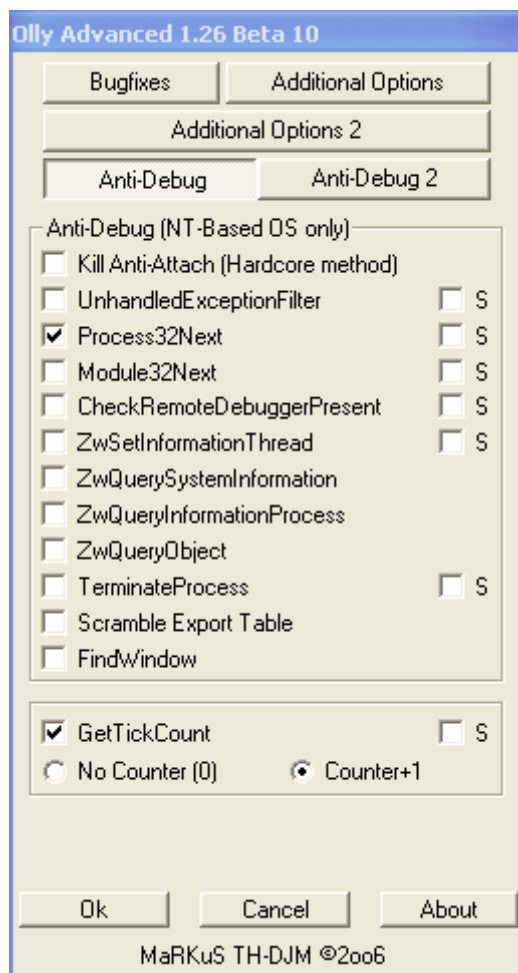
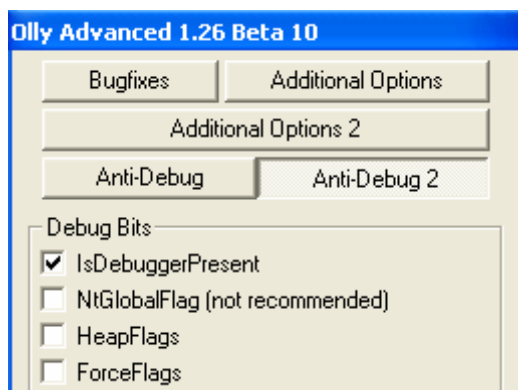


Figure 2.

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 one click. Of course we need to handle the IsDebuggerPresent API. We do so on the next tab (Anti-Debug 2) seen below:



PRIMER ON DUMPING ACTIVEMARK V6.X

Figure 3.

1.3.2 Checking out the target

We first open our target in Olly. Below is the [Memory Map] view of the target:

00400000	00001000	_kakuro		PE header	
00401000	00149000	_kakuro	.text	code	Code
0054A000	00023000	_kakuro	.rdata		
0056D000	00015000	_kakuro	.data	data	
00582000	00002000	_kakuro	.rsrc		
00584000	0000A000	_kakuro	.reloc		
0058E000	000B5000	_kakuro	.text		Level 3
00643000	0003C000	_kakuro	.data		
0067F000	00010000	_kakuro	.bss		
0068F000	00003000	_kakuro	.idata		
00692000	00003000	_kakuro	.rsrc		
00695000	000DC000	_kakuro	.text		Level 2
00771000	00001000	_kakuro	.idata		
00772000	00002000	_kakuro	.rsrc	resources	
00774000	00015000	_kakuro	.text	EXE	Level 1
00789000	00006000	_kakuro	.bss		
0078F000	00009000	_kakuro	.data		
00798000	00001000	_kakuro	.idata	imports	

Figure 4.

Notice that we have (4) distinctive layers:

1. Level 1. (the module's Entry Point). This level is responsible for general anti-debugging measures and the decompression / decryption of level 2.
2. Level 2. This level is responsible for the decompression / decryption of level 3, building the imports and the original code Sections. I should point out that decompression / decryption is not only code related, but also data, resources, imports, etc associated with a level or levels. Not to be confused with decrypting any external resources or data files which is a separate process. This is the level (or layer) that we will dump at. This is also the level (or layer) that we used to dump at in AM releases prior to v6.x.
3. Level 3. This level is responsible for the bulk of AM specific processing, anti-debugging, decryption and anti-dumping which used to occur in Level 2 in AM releases prior to v6.x
4. Original Code Section. Relax.... It isn't all original. AM has inserted code here (stolen API's) to jump back to its Level 3 or what I'll call the real Protection layer.

1.3.3 In the beginning...

Much has changed from previous releases. AM now utilizes some new features to discourage dumping and patching an application. These changes are not readily apparent until you reach or try and dump at Level 3.

Since this Tutorial is most concerned with dumping an application, we will concentrate on Level's 1 and 2.

Let's have a look:

LEVEL 1:

Level 1 is sneaky. 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. Level 1 now includes a DWORD value in the data section which is integral to Level 3 processing. This DWORD value is then stored in Virtual Memory via, you guessed it, A call to VirtualAlloc. In time, you'll be able to scan the .data section and find this hidden value, but we can simply put a

PRIMER ON DUMPING ACTIVEMARK V6.X

HWBP on the VirtualAlloc API. Before I forget, don't use Software BP's on this version. If you screw around with any code or API's, also, as in the past, close any third party tools while using Ollydbg or you may get the following:

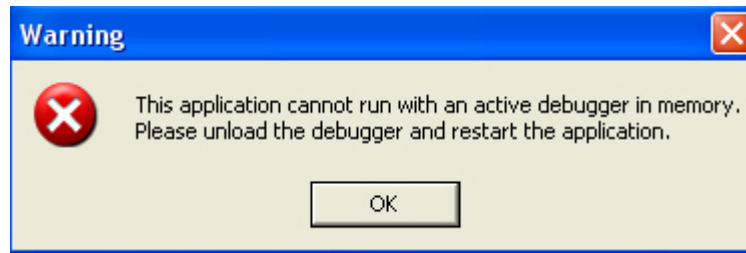


Figure 5.

You will need to learn how to best utilize your limited Hardware Breakpoints. We can start by setting our first HWBP on the VirtualAlloc API by using context help in Olly's [Executable modules) window for kernel32.dll:

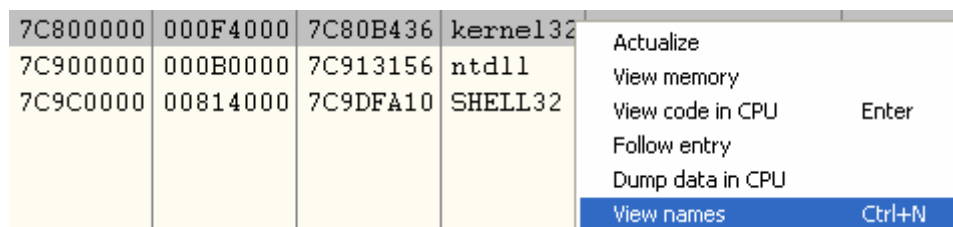


Figure 6.

Selecting View names then,

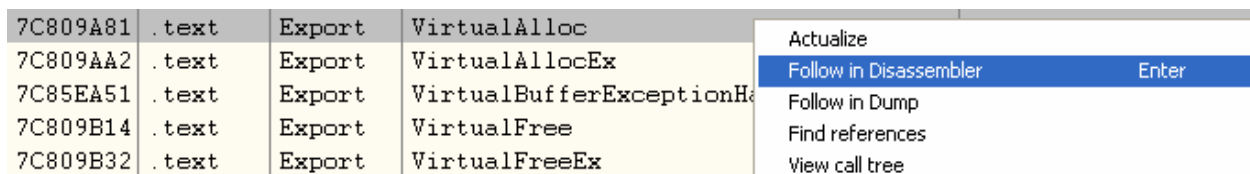


Figure 7.

Selecting VirtualAlloc and Follow in Disassembler and setting a HWBP on this function's beginning address for Hardware, on execution:

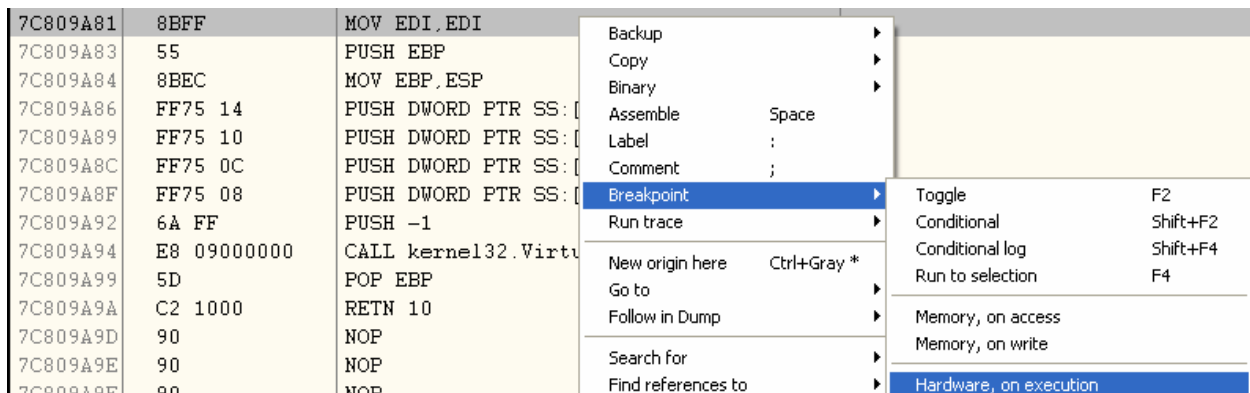


Figure 8.

AM doesn't perform any exception handling on HWBP's or CODE: 0x80000004.

We can now Run (F9) our target in Olly until we reach our HWBP.

Important Note: There are many breaks on VirtualAlloc, but we are only interested in one:

PRIMER ON DUMPING ACTIVEMARK V6.X


0012FFA8	00783027	CALL to VirtualAlloc from _kakuro.00783022
0012FFAC	00000000	Address = NULL
0012FFB0	00001000	Size = 1000 (4096.)  Look for this
0012FFB4	00001000	AllocationType = MEM_COMMIT
0012FFB8	00000004	Protect = PAGE_READWRITE
0012FFBC	0012FFE0	Pointer to next SEH record
0012FFC0	007740D3	SE handler
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C910738	ntdll.7C910738

Figure 9.

Yes, Size matters. We need the one for 1000 hex (4096 bytes). Follow the return above at address 00783022. Since we don't have a properly analyzed and configured PE file, we can do this by Execute till return in Olly (Ctrl+F9) to our return address. At this point, the only 2 register values we are interested in is EAX and EDI. We will now single step through the code until you reach the following:

00783037	A3 73F87800	MOV DWORD PTR DS:[78F873],EAX
0078303C	C700 85A85556	MOV DWORD PTR DS:[EAX],5655A885
00783042	EB 02	JMP SHORT _kakuro.00783046
00783044	83C8 BB	OR EAX,FFFFFFBB
00783047	0D 6FBD9A1	OR EAX,A1F9BD6F

Figure 10.

AM moves the Virtual Memory Address to a DWORD pointer address in the .data section. It then moves a DWORD value 0x5655A885 (it is unclear how this value is derived) to the Virtual memory address returned in VirtualAlloc. We can see this more clearly if we step through these next 2 instructions and look at our dump window:

Address	Hex dump	ASCII
0078F86B	00 00 00 00 85 A8 55 56UV
0078F873	00 00 79 01 00 00 00 00	..y
0078F87B	49 9E 35 0A 1B 2C EC A7	I5. .,is

Figure 11.

We see the DWORD value 0x5655A885 and also the Virtual memory address 0x01790000 above. (Note: this VM address may be different on your machine). The special DWORD value above is consistent for this target, but changes for each application. How important is this DWORD value? I won't show here, but set a HWBP for hardware, on access DWORD for this value in Virtual Memory and see the activity. Better yet, zero out this value in Virtual memory and see what happens. You'll get an Access Violation followed by Debugged Program was unable to process exception and termination of the application. This is what would happen if you dumped the target normally. To avoid this problem, the dumper tool, takes into account the VirtualAlloc API and when it breaks, it sets the value in register EAX (returned VirtualAlloc Memory address) to equal register EDI, which contains a DWORD pointer address close to the address 0x0078F873 above in Figure 11.

We continue single stepping (F7) through the code until we reach some code similar to the following:

PRIMER ON DUMPING ACTIVEMARK V6.X

00783129	A3 6FF87800	MOV DWORD PTR DS:[78F86F],EAX
0078312E	31C0	XOR EAX,EAX
00783130	0FA2	CPUID
00783132	8915 77F87800	MOV DWORD PTR DS:[78F877],EDX
00783138	A1 6FF87800	MOV EAX,DWORD PTR DS:[78F86F]
0078313D	EB 04	JMP SHORT _kakuro.00783143
0078313F	6A EB	PUSH -15

Figure 12.

I have highlighted the CPUID instruction. AM stores the value returned in register EDX to a .data section DWORD pointer. The saved CPUID information is generally, although not always, stored near the same address as noted in Figure 11 above. We know that AM ties your machine id and other related stuff when activating an application for your machine. (i.e. generating a valid license file). It would appear that CPUID can also be used to deter dumping an application since this value would be unique for every machine and the resultant dump could only be run on the machine that dumped the application. This technique is available for use, but does not seem to be enforced at the moment. In Level 3, there is code that checks the CPUID with the value stored above in address 0x0078F877.

LEVEL 2:

How best to reach it? One easy way is to set a HWBP, on execution for the VirtualProtect API. Use the procedure referenced above for VirtualAlloc to do this. When the HWBP is reached we can set a memory breakpoint on access for Level 2 as shown below:

00695000	000DC000	_kakuro	.text		
00771000	00001000	_kakuro	.idata		
00772000	00002000	_kakuro	.rsrc	resou	
00774000	00015000	_kakuro	.text	SFX	
00789000	00006000	_kakuro	.bss		
0078F000	00009000	_kakuro	.data		
00798000	00001000	_kakuro	.idata	impor	

Actualize
 Dump in CPU
 Dump
 Search Ctrl+B
 Set break-on-access F2
 Set memory breakpoint on access

Figure 13.

Run the target in Olly, (F9) cycling through the HWBP's until you break here:

006959C9	8925 10506900	MOV DWORD PTR DS:[695010],ESP
006959CF	66:60	PUSHAW
006959D1	8925 1C506900	MOV DWORD PTR DS:[69501C],ESP
006959D7	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
006959DD	A3 18506900	MOV DWORD PTR DS:[695018],EAX
006959E2	E8 CBF8FFFF	CALL _kakuro.006952B2

Figure 14.

Address 0x006959C9 is our Level 2 EP. This is where the new dumper program would dump the application, rebuilding the PE header for the new raw offsets, which are equivalent to the virtual offsets, then append the encrypted data (stub file) to the end of the dump. Note there is no Import rebuilding necessary at this point. Why? Because the Imports, along with the rest of the code are built later on (see below):

PRIMER ON DUMPING ACTIVEMARK V6.X

00695B52	FF15 18696900	CALL DWORD PTR DS:[696918]	kernel32.GetModuleHandleA
00695B58	50	PUSH EAX	
00695B59	E8 C4FEFFFF	CALL _kakuro.00695A22	
00695B5E	56	PUSH ESI	
00695B5F	FF15 28696900	CALL DWORD PTR DS:[696928]	kernel32.ExitProcess
00695B65	5E	POP ESI	
00695B66	C3	RETN	

Figure 15.

The address highlighted above is the call that leads to Import building, code, resource, data decompression / decryption / protection and Level 3 processing. If we follow that call we eventually land here (see below):

00695A53	E8 93FEFFFF	CALL _kakuro.006958EB	
00695A58	8945 EC	MOV DWORD PTR SS:[EBP-14],EAX	
00695A5B	837D EC 00	CMP DWORD PTR SS:[EBP-14],0	
00695A5F	74 1B	JE SHORT _kakuro.00695A7C	
00695A61	68 18516900	PUSH _kakuro.00695118	ASCII "Error executing the compression layer"
00695A66	68 D0686900	PUSH _kakuro.006968D0	
00695A6B	FF15 14696900	CALL DWORD PTR DS:[696914]	kernel32.lstrcpyA

Figure 16.

The call at address 0x00695A53 above is the critical step in the Level 3 and beyond building process. Notice the error message that follows if there's a problem. So, one might ask, why not trace further and find the Level 3 EP? There is no easy answer at this point. If we dump at Level 2 EP, the REAL code is not exposed and patching is difficult. If we dump at Level 3 EP, which is the REAL Level 3 EP? ActiveMark has imbedded the GetVersion and GetCommandLineA API's in obfuscated code prior to the next logical API, GetStartupInfoA. Or maybe the new logical Level 3 EP is as follows:

005D3AB2	6A 60	PUSH 60
005D3AB4	EB 03	JMP SHORT dumped1_.005D3AB9
005D3AB6	30F5	XOR CH,DH
005D3AB8	64:68 70856400	PUSH dumped1_.00648570

Figure 17.

After all this is a VC++ program. Or maybe we should dump at the very 1st instruction in Level 3 below:

005C658D	EB 02	JMP SHORT dumped1_.005C6591
005C658F	EB FF	JMP SHORT dumped1_.005C6590
005C6591	8925 B0CE6800	MOV DWORD PTR DS:[68CEB0],ESP
005C6597	60	PUSHAD
005C6598	8925 ACD76800	MOV DWORD PTR DS:[68D7AC],ESP
005C659E	EB 02	JMP SHORT dumped1_.005C65A2

Figure 18.

Looks innocent enough. We got familiar turf here. TEB stuff being moved to a data address before a PUSHAD.

For a variety of reasons, the targets tested did not run when dumped at these locations. Even after rebuilding Imports and appending the encrypted data... ActiveMark is using a marked number of saved pointers in its data sections and it seems that every dump will fail at similar locations regardless of the Level 3 EP chosen for each target. This is not to say it is not doable, but regardless of the method chosen, any dumped application provides significantly greater opportunities for analyzing and patching. The scope of this Tutorial is to highlight some of the obvious and not so obvious revelations of dumping the new release at Level 2 so that the reader can arrive at their own conclusions after further analysis.

PRIMER ON DUMPING ACTIVEMARK V6.X

We have one more thing to consider. ActiveMark has integrated the Level 1 EP as part of their processing. It is presumably to verify the integrity of the application (READ: it was not altered, dumped, or had sections removed). If we restart the application at Level 1 EP and view the [Memory map] window in Olly. Dump the PE Header section (1st one) and go to the address pictured below that contains the AddressOfEntryPoint. We want to set a HWBP, on access for this DWORD address.

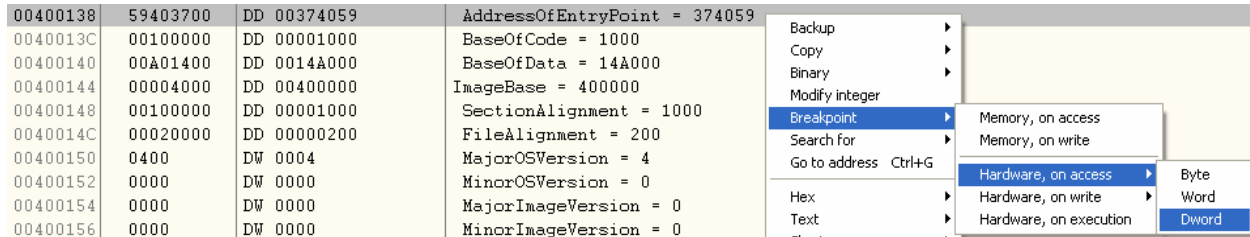


Figure 19.

Now Run (F9) the target process until we hit this HWBP and come to some code similar to the following:

005DC23B	5A	POP EDX
005DC23C	8B58 3C	MOV EBX,DWORD PTR DS:[EAX+3C]
005DC23F	01C3	ADD EBX,EAX
005DC241	0343 28	ADD EAX,DWORD PTR DS:[EBX+28]
005DC244	3950 F8	CMP DWORD PTR DS:[EAX-8],EDX
005DC247	74 12	JE SHORT _kakuro.005DC25B
005DC249	68 00000000	PUSH 0
005DC24E	E9 192CFFFF	JMP _kakuro.005CEE6C

Figure 20.

And a picture of the Registers as they exist at this HWBP:

Registers (FPU)		
EAX	00774059	OFFSET _kakuro.<Modu
ECX	0000000A	
EDX	87086EAF	
EBX	00400110	ASCII "PE"
ESP	0012F988	
EBP	0012FCE8	
ESI	00000105	
EDI	0068CF64	_kakuro.0068CF64
EIP	005DC244	_kakuro.005DC244

Figure 21.

Notice the value in register EAX. It's our Level 1 EP. And one more picture of the code pane showing the respective values at EIP 0x005DC244:

```
EDX=87086EAF
DS:[00774051]=87086EAF
```

Figure 22.

PRIMER ON DUMPING ACTIVEMARK V6.X

If we were to have changed the EP (due to dumping, alteration, etc.) Do you think the value in register EDX would equal the value in DS: [00774051] ? Probably not and the program would terminate via ExitProcess. To effectively counter this verification process a simple change to the Level 1 EP address's original instruction(s) is required. We can do the following to the original EP:

00774059	68 C9596900	PUSH dumped.006959C9
0077405E	C3	RETN
0077405F	9C	PUSHFD
00774060	90	NOP

Figure 23.

Replace the original instructions with a PUSH to our Level 2 EP. (or your chosen new EP) and RETN. The dumper tool does this automatically, but I reference it here for the manual dumping approach. You'll need to save all your changes before dumping with your favorite PE dumping tool. No need to use ImpRec or rebuilding the imports. Then you would append the encrypted data to the end of your dump with a hex editor and your dump should run cleanly, like the original program. Or use the dumper tool which does this all automatically.

1.3.4 Analyzing the target

This section is a reminder that after dumping any ActiveMark application, it is always a good idea to modify the BaseOfCode section in the PE header. This way you can better analyze the code and literals referenced in Level 3. To do this, simply open your dumped file in LordPe or similar and make the following change highlighted below:

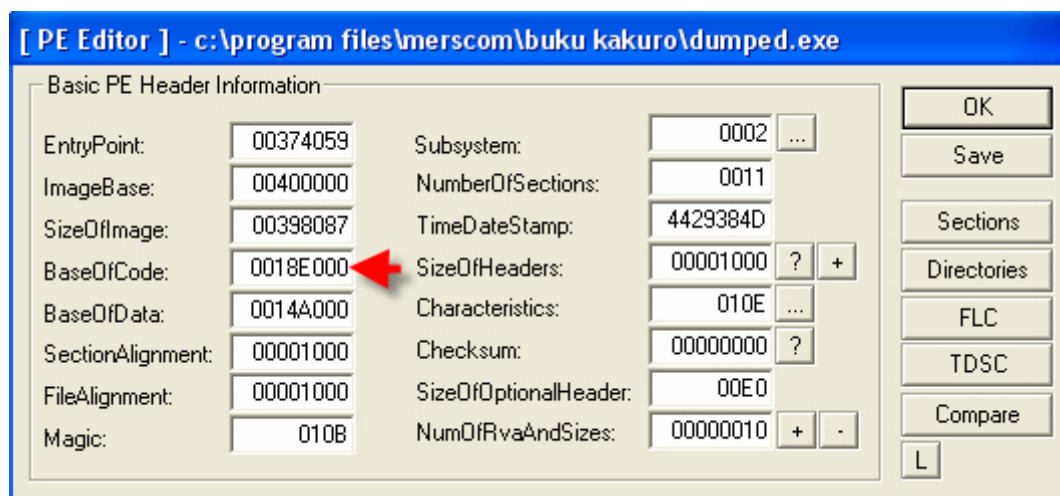


Figure 24.

1.3.5 The AMDUMPV6 (Dumper Tool)

This tool is basically meant to show a very basic way in which to dump applications using the new AM v6.x protection system. As outlined in this Tutorial, the tool will dump an application at Level 2. It will take into account the DWORD value moved to Virtual Memory and reference that data within the .data section it resides. It will modify the original EP instructions to point to the new Level 2 EP. It will dump the process, fixing the raw offsets to make a valid PE file. Finally, it will append the encrypted data to the end of the dump and allow the reader to save the resultant dump file, which can then be executed as the original.

The full source is included with the contents of this Tutorial. Numerous comments are interspersed with the "C" language code. The reader can see the flow of how the process is created, anti-debugging measures handled, hardware breakpoints set and cleared, single stepping breakpoint exceptions handled and finally, how another running process can be read into the VM of the calling process, how to apply the PE header structure to this Virtual

PRIMER ON DUMPING ACTIVEMARK V6.X

Memory that contains the full dumped target executable, how to fix the raw offsets in the PE header, and how to append the encrypted data file of the original executable to our dumped file.

1.4. References

The target is a game called Buku kakuro - 31.3 Mb - 6.1.342 AM release. You can get it at the link below:

http://d.trymedia.com/dd/merscom/60m_d/merscom/BukuKakuroSetup.exe

Other references can be found in the AMDUMPV6 source code as applicable.

1.5. Conclusions

There are many new features in ActiveMark v6.x to learn and control. This Tutorial is meant as a learning tool to demonstrate the most basic method to dump an AM v6.x application. If, after reading this Tutorial, better methods in dumping AM v6.x applications can be realized, then the author has accomplished his goal. There are still numerous obstacles to overcome after dumping an application. Some of these obstacles will be addressed in Part II – Loading, patching and running an application under AM v6.x.

All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.

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/>