

ArmaGeddon v1.0 – Conceptual overview on tool for unpacking Armadillo

Version 1.0
February 2008

1. Forewords

The SoftwarePassport/Armadillo Software Protection System has been with us now for some time. During this time, many Tutorials, forum posts, scripts, tools and other ideas have been formulated and exchanged with one singular purpose. How to best remove, circumvent or otherwise disable the armored shell and reveal the naked animal. I think AndreaGeddon's document, "Armadillo 4.20: Removing the armour: a naked animal" concerning this subject (more than 2 years old as of this writing) set the bar on revealing some interesting details on the Armadillo protection system and how best to reverse it. There are other Tutorials of equal significance that showed us the paths to unpack or deal with various aspects of the protection system, but not mentioned here. Various tools have been developed to make our job somewhat easier in reversing, unwrapping, unprotecting, defeating and well you get the idea. Tools are born out of necessity, to make our lives easier, to reduce the amount of work (redundancies) and effort to get a job done, etc... I applaud the efforts of those who have written such tools and Tutorials to benefit the RCE community at large. Tutorials provide a wealth of information to this end. Tools can have interesting names such as this one.

Dictionary.com reveals this about Armageddon:

Ar·ma·ged·don Pronunciation [ahr-muh-ged-n] -noun

1. the place where the final battle will be fought between the forces of good and evil (probably so called in reference to the battlefield of Megiddo. Rev. 16:16).
2. the last and completely destructive battle: The arms race can lead to Armageddon.
3. any great and crucial conflict.

I think you will agree on at least one of the above explanations. Read on to learn more about this tool and some of the "Best Practices" that went into its development.

*Have phun,
CondZero*

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 damages 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. Conceptual overview of ArmaGeddon v1.0 tool: Unpacking Armadillo	3
1.1. Abstract	3
1.2. Targets	3
1.3. Limitations & Features	3
1.4. The Big Picture – “It’s so easy, a caveman can do it”	4
1.5. Nuts & Bolts	5
1.5.1 The Debug engine	6
1.5.2 The Search Engine	9
1.5.3 Other components of note include:	10
Flags Register	11
Status Flags	11
1.5.4 The Road to Glory	12
1.6. References	18
1.7. Conclusions	18
1.8. Greetings	18
Document History	18

1. Conceptual overview of ArmaGeddon v1.0 tool: Unpacking Armadillo

1.1. Abstract

Quite a bit of testing goes into creating tools (programs) which can perform units of work correctly and efficiently. It is not always easy to create generic programs which can work on more than a couple of different OS's or versions of a particular software product. To make these programs flexible, maintainable, reliable and generic takes a considerable amount of time, patience, effort and skill. Developers have time constraints, learning curves, continuous adjustments and other real life or imagined considerations when designing and creating a program. I realize that the temptation to distribute or release something too quickly, to get it out there, to be known, etc... is very tempting indeed. It is also very frustrating to use such programs when they haven't gone through any perceptible quality control. This project is the culmination of more than a few people's contributions and efforts. Also important, this project brings together some of what I feel are "Best Practices" both in the integration and utilization of other people's works (imbedded into the project) and proven methodologies that can be programmatically created. The basis of this Tool is "The Possibilities" in eliminating the need for other Tools of this type. While that is not always possible, ArmaGeddon does a pretty good job of keeping our toolbox manageable.

1.2. Targets

SoftwarePassport/Armadillo Software Protection System Version 5.40 (Public Build) 32-bit Trial Edition (emulating professional) available from:

http://www.siliconrealms.com/armadillo_engine.shtml

This tool should become available on the Arteam's releases / tools section:

<http://arteam.accessroot.com/releases.html>

1.3. Limitations & Features

Armageddon has been tested on various targets ranging from versions 3.78 through and including 5.40 under Win2K, WinXP and Vista 32 bit (even on x64). If you experience any problems running the program, you may need to download and install Microsoft Visual C++ 2005 Redistributable Package (x86) available here:

<http://www.microsoft.com/downloads/details.aspx?familyid=32bc1bee-a3f9-4c13-9c99-220b62a191ee&displaylang=en>

1.3.1.1.1 Brief Description

The Microsoft Visual C++ 2005 Redistributable Package (x86) installs runtime components of Visual C++ Libraries required to run applications developed with Visual C++ on a computer that does not have Visual C++ 2005 installed.

ArmaGeddon attempts to accomplish the following:

- User friendly GUI interface that is intuitive in design
- Minimal options and intervention required in reversing
- Minimize speed, maximize versatility
- Minimum and standard protections supported
- Debug-Blocker
- Copymem II
- Nanomites
- Import table elimination
- Import redirection
- Code Splicing

- Memory patching
- Dumping a process
- Rebuilding imports

The program does not attempt to remove unwanted sections of a program. Dll's are currently not supported. Targets that utilize overlays or Shockwave Flash are not supported. As to are various other features of the Armadillo protection system, i.e. your application has expired, uses hardware locking, requires a key to use or integrates custom protections and secured sections. To the extent possible, the program can navigate to the OEP for applications that have been compressed prior to being protected by Armadillo i.e. UPX. ArmaGeddon is also dependent on a PE structure (i.e. section names) that conforms to standard armadillo naming conventions:

```
entry point = 004c9000
From      - To      Section  Size
-----
00400000 - 00401000  PE       1000
00401000 - 00448000  .text    47000
00448000 - 0044B000  .rdata   3000
0044B000 - 00479000  .data    2E000
00479000 - 004C9000  .text1   50000
004C9000 - 004D9000  .adata   10000
004D9000 - 004F9000  .data1   20000
004F9000 - 00639000  .pdata   140000
00639000 - 007D0000  .rsrc    18E000
```

and not...

```
entry point = 004c9000
From      - To      Section  Size
-----
00400000 - 00401000  PE       1000
00401000 - 00448000  .xxyy    47000
00448000 - 0044B000  .aabbcc  3000
0044B000 - 00479000  .ddff    2E000
00479000 - 004C9000  .gghhi   50000
004C9000 - 004D9000  .jjkkll  10000
004D9000 - 004F9000  .mmnnpp  20000
004F9000 - 00639000  .qrrss   140000
00639000 - 007D0000  .ttuu    18E000
```

Although we can easily determine the 1st section as being text / code, some applications make use of .idata sections, import sections, etc. depending on protection options and physically renaming or assigning their own section names can cause problems.

1.4. The Big Picture – “It’s so easy, a caveman can do it”

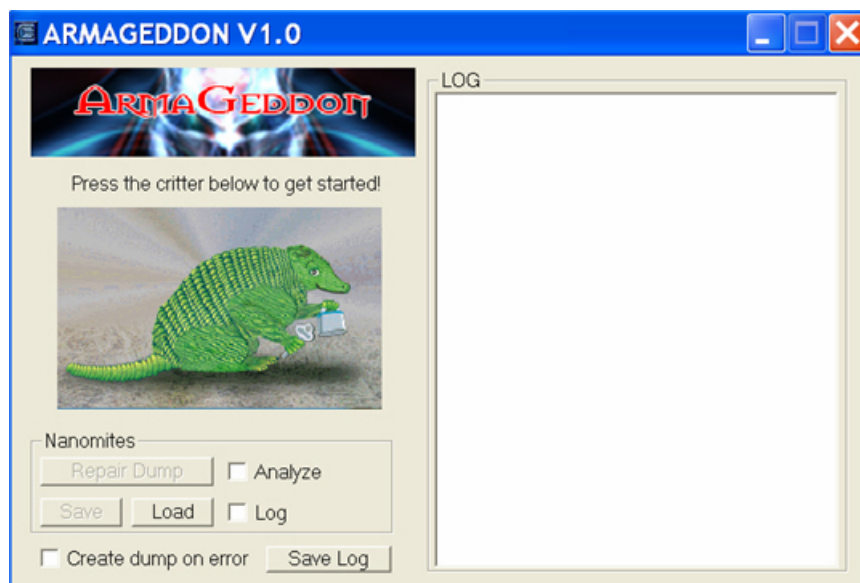


Figure 1

The main GUI screen above is relatively simple and straightforward. The top bitmap “ArmaGeddon” when clicked will give you the “Credits”. The “Armadillo” bitmap will invoke launching and reversing the target application. The “Log” will capture and optionally allow you to save all information recorded during the reversing operation to a file named “Armageddon_log.txt” in the target application’s folder. For nanomites, we have the same basic options as are found in the Arminline tool from Admiral. Note: The same basic processing as well. The ability to load a saved *.nan file and repair a dump. The ability to save an analyzed nanomite table to a *.nan file and use later. The “Analyze” checkbox will automatically scan for INT3 “0xCC” bytes and process them in the same fashion as Admiral describes in his readme.txt file. A “Log” checkbox will display all INT3 instructions encountered “real time” during the course of using the target which can be saved and used to analyze later. The program will automatically ask you to dump the file when completed to a “dumped” file. You have the option of dumping the file yourself using any 3rd party tool i.e. LordPE or similar. If you suspect that nanomites are present, you should now, if not already done so, check the analyze box. After pressing “OK” on the dump, Armageddon will automatically search for a valid IAT and all respective “valid” imports (no matter how shuffled they may be) and rebuild the IAT and add a new section to your dump file with a “_” appended to your dumped file name:

i.e. dumped.exe >> dumped_.exe

where: dumped.exe is the initial saved dump and dumped_.exe is the IAT rebuilt dump.

After doing this, if any nanomites are present, the external nanolib.dll will be invoked to handle any INT3 instructions found in the child process, brute forcing the process to build a “condition table” of values and ultimately arrive at the proper jump sequences. When completed, you are now able to press the “repair dump” button to fix your IAT dumped file to: >> dumped_ NanoFix.exe. This appends Admiral’s nano handler and final nano table to the dumped file.

In some cases, the nanomites may be imbedded in certain functions that are not utilized when the application loads. In these cases, you will specifically have to find one instance (i.e. use a function with imbedded INT3 instructions) of a nanomite to initiate the process.

1.5. Nuts & Bolts

Everybody wants to know “what’s under the hood”.

Armageddon was built using VC++ in the VS2005 environment / WinXP OS and includes 2 major components:

- Core debugging engine
- Core search engine

Complimentary components include author / component:

- Nacho_dj: ARTeam Import Reconstructor 1.0 (Brand new) (imbedded)
- Admiral: Nanomite processing engine (used in Arminline tool) Nanolib.dll adapted to this program
- Z0MBiE: XDE v1.02 eXtended length disassembler engine based on LDE/ADE engines (imbedded)

Anyone that has programmed an unpacker, dumper, memory patcher (loader) or similar tool is familiar with the standard debugging loop:

We normally launch the target application via:

- CreateProcess API with creation flags set to either DEBUG_PROCESS (both parent & any spawned children) or DEBUG_ONLY_THIS_PROCESS (single process).

We handle any anti-debugging tricks via:

- Internal functions like hide_debugger
- Set our Software breakpoints on the 3rd, 4th, 5th byte of an API to avoid memory checks on API’s
- Use hardware breakpoints and single stepping

We wait on debugging events to occur via:

- WaitForDebugEvent(&DebugEv, dwTime) where the structure DEBUG_EVENT gets loaded (in this case to our pointer address &DebugEv) with exception information and we designate some amount of time dwTime to wait on the event. Normally, dwTime would be set to 1000 milliseconds or 1 second.

We handle the debugging events that we are interested in via:

- case EXCEPTION_DEBUG_EVENT:

Depending on the protections used by a target, we would be interested in the following:

- case EXCEPTION_BREAKPOINT: - Software “0xCC” breakpoints, not only ours, but in the case of nanomites, the child process (0x80000003)
- case EXCEPTION_PRIV_INSTRUCTION: - Virtually every Armadillo target exhibits this exception from the virtual armadillo dll (0x80000096)
- case EXCEPTION_GUARD_PAGE: - used for copymem 2 and finding the OEP (0x80000001)
- case EXCEPTION_SINGLE_STEP: - used for hardware breakpoints and single stepping (0x80000004)
- case CREATE_PROCESS_DEBUG_EVENT: - used for determining debug-blocker

1.5.1 The Debug engine

I don’t believe in reinventing the wheel, if something else works, use it (credits where credits are due of course). I will also talk about “Best Practices”. What do we mean by this? Over time, certain methods and principles are adopted by virtue of their ease of use, practicality and proven methodology. In order to be generic, and support all protection features, some generally accepted ideas will not work within the context of a live program being analyzed and reversed by an unpacker. In the case of Armadillo, we must maintain the relationship between the father and child process if both exist.

Let’s explore this mysterious “Best Practices”. In an ideal world, we have teams of developers and testers and people with a lot of bucks to pay us to go on. In the RCE world, we have you, the reader, who constantly challenges one’s self to learn a protection and apply what they’ve learned. Probably doesn’t have much money to spend on software, but will willingly spend whatever they have on a good piece of “code”.

Armageddon was designed to **not** fight with the Armadillo protection system, but to embrace the protection. It will gladly accept “Debug-Blocker”, “Copymem 2” and “Nanomites” without protest. Why? Because if you look at the 1st point, the CreateProcess API uses the creation flags DEBUG_ONLY_THIS_PROCESS. That doesn’t seem right. How do we detach from the father and attach to the son? We don’t. Within the main debug loop (i.e. the father) we set a software breakpoint on the return (that’s RETN) of the WaitForDebugEvent API. (Note: We do this for other API’s as well. We will need them during the course of reversing / unpacking). What this does for us is it allows us to capture all the debug events of the child process without specifically having the debug api’s do it directly. In this fashion, we have **almost** the same level of communication as we do with the regular debugging api. Each and every time the child process fires a debugging event, we capture it by resetting our software breakpoint (via single stepping) on our inner father WaitForDebugEvent API and interrogating the DEBUG_EVENT structure returned from the child process. Most importantly, the father process handles the exceptions for copymem-II and nanomite processing and the subsequent ContinueDebugEvent automatically. We don’t, however, have the use of hardware breakpoints or single stepping of the child. For this I chose ZOMBiE’s: XDE v1.02 eXtended length disassembler engine, based on LDE/ADE engines. This fantastic tool allows us to either assemble an instruction or, as in the case of this program, disassemble an instruction “on the fly” into its length and opcode. We use this for single stepping through instructions. i.e. we add the length of the next instruction (pointer address) to our current Context.Eip value. We are also invisible (as far as the child is concerned) from being detected as a debugger.

The prototype:

```
/* returns:
*/
/* 0 if error
*/
/* instruction length (== diza.len) if OK */
/* prototypes used in XDE v1.02 eXtended length disassembler engine
*/
int __cdecl xde_disasm(/* IN */ unsigned char *opcode,
```

```

        /* OUT */ struct xde_instr *diza);

int __cdecl xde_asm(/* OUT */ unsigned char* opcode,
                  /* IN */ struct xde_instr* diza);

```

I won't show the structure of the XDE engine but the CALL looks like this:

```

// Perform the XDE length function
dwXDEreturn = xde_disasm((unsigned char *)dwXDEAddress,&diza);

```

dwXDEreturn is our DWORD return length of the function. We pass a pointer to our address in memory and a pointer to the structure used by the function (&diza). The function will fill the structure and return the length.

We would then do something like this to set a pseudo single step in our code:

```
SWBPEXceptionAddress[8] = (DWORD)CebugEv.u.Exception.ExceptionRecord.ExceptionAddress+dwXDEreturn;
```

We set a software breakpoint (SWBPEXceptionAddress[8] a DWORD array) and assign it the value of our current exception address + the value returned by the XDE engine disassemble function.

A couple of points. First, we have established all necessary handles to the child process and thread by doing the following (having adjusted our privileges in advance):

```

childhProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE,CebugEv.dwProcessId);
childhThread = OpenThread(THREAD_ALL_ACCESS,FALSE,CebugEv.dwThreadId);

```

This is done in the `CREATE_PROCESS_DEBUG_EVENT`.

Note the last arguments in the called functions above: `CebugEv.dwProcessId` & `CebugEv.dwThreadId`. We have captured the debug event information in a `DEBUG_EVENT` structure for the child process. We have all the information we need to process any specific debug event as we would in a normal debug loop.

We setup our case structures and switches accordingly just as we do in a main debug loop. A loop within a loop.

Second, we have established our debugging loops and environment and have set all necessary breakpoints or stored the breakpoints (API specific) in advance to use them later. We use API's, just as the Tutorials suggest, to establish a means of "just in time" applying breakpoints and doing specific processing "while the code is decrypted and available" to use because Armadillo does a good job of decrypting code "on the fly" just when it needs it and then reusing the code blocks in memory for something else.

Third, for single process armadilloes, the steps involved are somewhat easier. The magic of the Armageddon program is that it doesn't care whether the target is a single or dual process and has been setup to handle either condition automatically. This also allows the running process(es) to run unabated with a minimal amount of intervention. This is what is necessary to perform Copymem II and nanomite processing. We also have the capacity as you will see later to enable logging of nanomite processing in a target "on the fly" and perhaps use this information to diagnose any nanomites that could be causing problems in a rebuilt dump.

Let's have a look at the `DEBUG_EVENT` structure:

```

typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
    };
};

```

```

        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;

```

We are mainly interested in `dwDebugEventCode == '1'`.

The union of structures (Event codes) is numbered from '1' to '9' with '1' being the 1st `EXCEPTION_DEBUG_INFO` Exception; and '9' being the last `RIP_INFO` `RipInfo`.

`dwProcessId` was used to establish our handle for the child.
`dwThreadId` the same, very useful when manipulating thread contexts.

Once we have established the above, we can then drill down to our respective INFO structures, the most important of which, is `EXCEPTION_DEBUG_INFO` Exception:

```

typedef struct _EXCEPTION_DEBUG_INFO {
    EXCEPTION_RECORD ExceptionRecord;
    DWORD dwFirstChance;
} EXCEPTION_DEBUG_INFO, *LPEXCEPTION_DEBUG_INFO;

```

and finally to this:

```

typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    DWORD NumberParameters;
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;

```

The Exception code (not to be confused with Event code) is the `0x80000003` `DWORD` value associated with say an `EXCEPTION_BREAKPOINT`. The `ExceptionAddress` is equally important and used to determine **which** API or armadillo code issued the exception.

By having the structures in our program, we don't have to deal with stack pointers and the like. We can specifically point to information directly in our structures and read / write / manipulate it at will. Then it just becomes the usual `ReadProcessMemory` or `WriteProcessMemory` (the process...) to resolve process memory manipulation. For the same reasons, we have the `CONTEXT` structure:

```

/* Cut */
//
// This section is specified/returned if the
// ContextFlags word contains the flag CONTEXT_INTEGER.
//
DWORD   Edi;
DWORD   Esi;
DWORD   Ebx;
DWORD   Edx;
DWORD   Ecx;
DWORD   Eax;
//
// This section is specified/returned if the
// ContextFlags word contains the flag CONTEXT_CONTROL.
//
DWORD   Ebp;
DWORD   Eip;
DWORD   SegCs;

```

// MUST BE SANITIZED

```

DWORD   EFlags;           // MUST BE SANITIZED
DWORD   Esp;
DWORD   SegSs;
/* Cut */

```

We can interrogate and manipulate our Registers (i.e. Eax, Ecx, etc.) our Eip for single stepping / resetting breakpoints, etc. EFlags becomes important when dealing with nanomite processing and determining jump conditions. We have this information for both father and child (dual process i.e. debug blocker). This program uses the EAX register to interrogate the values returned by the VirtualAlloc API used in allocating the virtual armadillo dll to VM, also for determining Code Splicing.

1.5.2 The Search Engine

Many times it is necessary to find bytes of code “hex strings” in memory that require modification or can be used to navigate further to variables and associative (data values) or step (trace) into additional code. This program makes use of a robust Boyer-Moore search string algorithm with wildcard support to find the needed hex strings (which are converted to ascii strings) for searching. The following hex strings are used:

```

//***** SEARCH STRING ARRAYS *****/
//SWBP'S ON API RETN INSTRUCTION
//Note: hex "3F" denotes wildcard string "?"
char *hexapiiretn[1]={ "C23F00" };
//Armadillo virtual memory code specific hex search strings & .text section
char *hexarmcode[4]={ "3B153F3F3F3F0F8E", "8B853F3F3F3F2B853F3F3F3FC1E802",
"6800010000", "E83F3F3F3F83C40C" };
// INDEX [0] = COPYMEM2 HEX SEARCH STRING
// INDEX [1] = IAT ELIMINATION HEX SEARCH STRING
// INDEX [2] = IAT REDIRECTION HEX SEARCH STRING (PUSH 100)
// INDEX [3] = IAT REDIRECTION HEX SEARCH STRING (CALL ????????, ADD ESP,0C)

```

The above search string arrays can be defined in advance and are utilized when needed. You will notice the API array which includes a single search string. All the API returns take on the form RETN instruction “C2??00”. The hexarmcode hex strings are used for searching the virtual armadillo dll with the exception of the 1st which searches the .text1 armadillo pe section. A 3rd hex string is built and called dynamically as the variable is not known at run time. We will see how each is used in this program. There are a couple of things to consider.

- All searches require that we copy via ReadProcessMemory API the required memory block(s) from the process we are interested in into our current process's memory first via VirtualAlloc API, before conducting the search. We are not using dll injection or CreateRemoteThread API to do this, although we could.
- The assumption is made that all the required system dll's in our target application's process will load in the same addresses as our current process.

The overhead is minimal and the sub second times barely noticeable during the course of searching.

As most of you readers are well aware, there are general weaknesses with the use of search strings. A simple code rewrite, _asm insertion or code obfuscation can easily render them invalid. One must also choose their search strings wisely to anticipate change and not use a reference to a particular code address as in the following:

```

004E9AA1    60                PUSHAD
004E9AA2   33C9             XOR ECX, ECX
004E9AA4   75 02            JNZ SHORT Armadill.004E9AA8    << here
004E9AA6   EB 15            JMP SHORT Armadill.004E9ABD    << here
004E9AA8   EB 33            JMP SHORT Armadill.004E9ADD    << here

```

The above are definitely not good candidates, nor is the following:

```

004E9B43   2B4D 0C          SUB ECX, DWORD PTR SS:[EBP+C]
004E9B46   8B56 78          MOV EDX, DWORD PTR DS:[ESI+78]
004E9B49   0355 08          ADD EDX, DWORD PTR SS:[EBP+8]

```

```

004E9B4C      8B5A 20      MOV EBX,DWORD PTR DS:[EDX+20]
004E9B4F      035D 08      ADD EBX,DWORD PTR SS:[EBP+8]

```

The register variables can change i.e. EDX could become ECX in the next release as to the displacement values can and do change i.e. [EDX+20] could become [EDX+24]. Static code is our only option and this too has limitations over time. For this reason, a checkbox option has been included with the program (see figure 1 above) “Create dump on error”. This allows you, the reader, to have an integral, proactive part in resolving errors and posting them to a forum perhaps for resolution later. The ArmaGeddon program will take a snap dump of the process, saving the virtual armadillo dll in VM to the .pdata section of the saved dump file. The original OEP will be intact. After this, one can use a debugger “Olly” to open the saved dump and navigate to the .pdata or .text1 section (code splicing) to find why a search string failed.

The MessageBox will give you the information you need before dumping, i.e. the hex string value and condition which failed.

1.5.3 Other components of note include:

ARTeam Import Reconstructor 1.0 (ARImpRec.dll)

Nacho_dj’s fabulous new Import Reconstructor tool v1.0. Our prototype used in this program follows:

```

/* prototype used in ImportReconstructor.dll */
extern "C" {
long __cdeclspec(dllimport) __stdcall SearchAndRebuildImports(DWORD
IRProcessId,LPSTR IRNameOfDumped,DWORD IROEP,DWORD *IRIATRVA,DWORD
*IRIATSize,DWORD *IRWarning);
}

```

I used the magical “**DLL2LIB**” evaluation program to (and these are not my words, but the author of DLL2LIB’s):

“convert a DLL file into its equivalent static library file. After that, you can replace the original DLL with the static library file, rebuild your application, and distribute it without the DLL file! The most exciting thing is that the conversion process DOES NOT require any source codes of the DLL files! All works are done from binary to binary. DLL to Lib will rebuild programming interface identical to the functions exported by the DLL and reconstruct the necessary symbol tables, string tables and reference tables from the DLL to make a valid static library for you!”

The SearchAndRebuildImports function takes 6 arguments. We only need supply the PID (process id of target),a pointer to the name of our dumped file IRNameOfDumped and the OEP IROEP. The function will automatically fill in the remaining arguments IRIATRVA (IAT start address), IRIATSize (IAT size) and a pointer to a warning literal value which can be associated with the actual return code itself. Let’s see how it’s called in the program:

Our function:

```

// Rebuild imports on dumped file
long DoSearchAndRebuildImports(DWORD thispid)
{
    IRINT = 0;
    IRINT = SearchAndRebuildImports(thispid, (LPSTR) savebuffer,OEPAddress,
        IRIatrva,IRIatsize,&IRwarn);
    return (IRINT);
}

```

The CALL:

```
IRretn = DoSearchAndRebuildImports(childpid);
```

We pass the process id of our target application. We expect a long (signed DWORD) as our return value. ‘0’ = Successful. The log will show the results and any error messages. Coded in Delphi. It performs the task of

rebuilding the import table in a new section. Its main feature is it can rebuild a shuffled IAT. Of note, I experimented with different solutions, but found this one to be the fastest and most reliable. “Best Practices”.

Nanomites:

I will show you some information (for the curious) as regards EFlags and the bits and bytes of what determines how a nanomite is processed. There are Tutorials and tools that deal specifically on this subject not discussed here:

Flags Register

The flags register is a 32-bit register named **EFLAGS**. Figure 2-8 defines the bits within this register. The flags control certain operations and indicate the status of the 80386.

The low-order 16 bits of **EFLAGS** is named **FLAGS** and can be treated as a unit. This feature is useful when executing 8086 and 80286 code, because this part of **EFLAGS** is identical to the **FLAGS** register of the 8086 and the 80286.

The flags may be considered in three groups: the status flags, the control flags, and the systems flags. Discussion of the systems flags is delayed until Part II.

Figure 2-8. EFLAGS Register

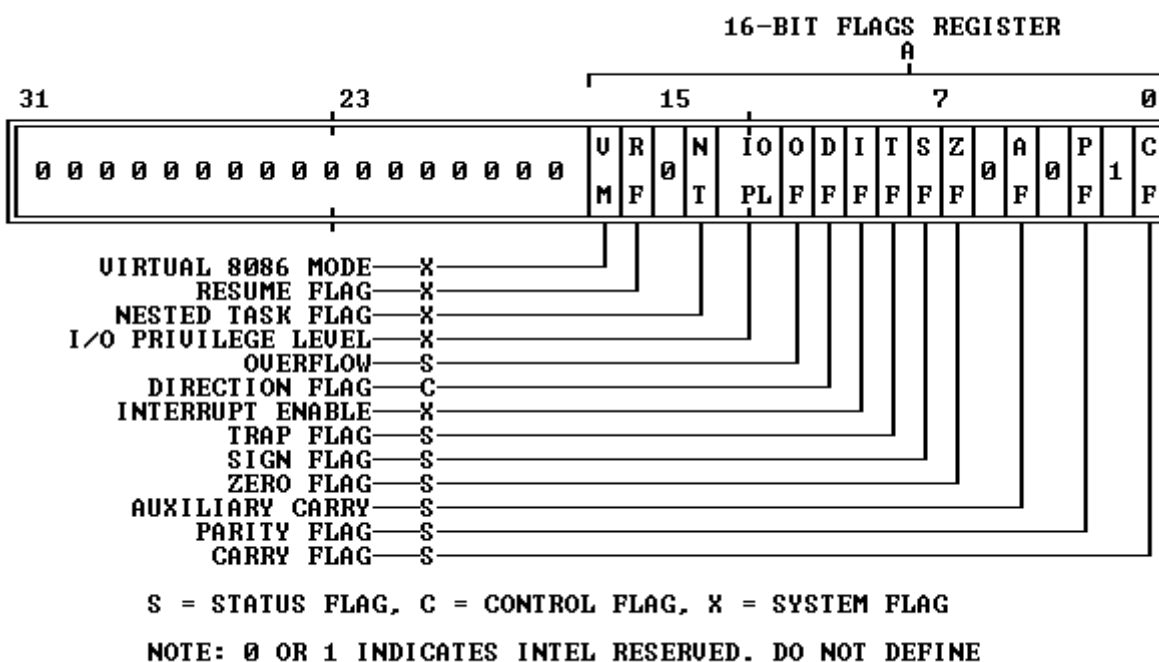


Figure 2

Status Flags

The status flags of the **EFLAGS** register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, **ZF**, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use **ZF** to signal that their operations are complete. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction. Refer to Appendix C for definition of each status flag.

Admiral's Nanolib source code, which is freely available from his site referenced later, provides the means to communicate with the nanolib.dll externally and has been adapted and integrated into ArmaGeddon. There are other views on nanomite processing that include extracting the nano tables and processing to produce external tables and

patch a target that way. I believe that Admiral's approach is the "Best Practices" way to resolve a very difficult problem. Disbelievers are encouraged to read the limitations and features once again.

1.5.4 The Road to Glory

... Contains many potholes. We've talked about the components and how they are integrated into Armageddon. Let's talk about the process of finding our way through the reversing jungle.

Once we have established the base address of the virtual armadillo dll and know whether we are dealing with one process or two (debug-blocker) we have certain choices that are dictated.

I won't go into customized protections as they are beyond the scope of this document.

With a single process, the protection features are limited to IAT redirection, IAT elimination, Code splicing and the like. With a dual process, we can add copymem II and nanomites to the mix.

We use the privileged instruction exception to establish:

```
// Set a SWBP on VirtualProtect API
// Set SWBP on VirtualAlloc RETN address
```

Why? We will be using these for what follows.

1.5.4.1 Code Splicing

Once the virtual armadillo dll is semi decrypted in memory, the above API's can point us to some vulnerable code. It is at this time that we can go onto the process of determining if code splicing is enabled. We do this by calling a function:

Determine_Strategic_Code_Splicing(HANDLE thisProcess, HANDLE thisThread)

Here we interrogate the arguments (stack values) returned by the VirtualAlloc API. It must meet the following requirements: [ESP+ decimal offset] used:

- EAX must return a value (generally called the hi-mem address)[Esp+4]
- Size must be greater than a predetermined value [Esp+8]
- Type must be MEM_RESERVE [Esp+12]
- Protect option must be PAGE_EXECUTE_READWRITE [Esp+16]

Are the above constraints always true? My testing revealed that we cannot expect a hi-mem value to be used by code splicing, nor can we expect the size to always be close to hex: 20000 bytes. This is a source of confusion and errors.

And another function if the above holds true:

Verify_Strategic_Code_Splicing(HANDLE thisProcess, HANDLE thisThread)

On the 2nd pass of VirtualAlloc, we interrogate (verify) the returned arguments (stack values) once again:

- EAX contains same value as previous
- Size contains same value as previous
- Type contains MEM_COMMIT (are we committed?)
- Protect contains PAGE_EXECUTE_READWRITE

Armageddon takes a proactive approach, as some tutorials suggest, and supplies an address value in Register EAX upon the 1st return described above see code below:

```
// Return [Section] of our choice in Register EAX return value
// 1st choice is .adata section
// Insure we have read/write to the section
if (!VirtualProtectEx(thisProcess, (LPVOID) AdataVMAddress,
```

```

        AdataVMsize, PERWProtect, &PEOldProtect))
{
    // Produce an error message and Return
}

```

The tutorials tell us that the .adata section may best serve as the most suitable destination for redirecting code splicing. Tests seemed to concur.

We make sure we have read / write set to this section before continuing. Note: if the size of the code splicing memory block is greater than the size of .adata a problem may occur. Unfortunately (at least in the newer variants of armadillo) there isn't the capacity to add contiguous sections of memory to the process for redirecting code splicing. Something to consider. Okay, I may have broken one of my rules here ☺.

1.5.4.2 Import Redirection / Elimination

When we reach our software breakpoint on the VirtualProtect API, we have some choices. We want to slowly find our way to the armadillo code which redirects / eliminates imports. At this point, this code has not been fully decrypted by the armadillo layers. We can possibly set a software breakpoint on the LoadLibrary API. What I have done is:

```
// Set a SWBP on GetModuleFileNameA RETN address
```

Once I arrive at this breakpoint, I can now begin to use the search engine to find some necessary hex search bytes. One thing to note. Before we do any memory searches against the virtual armadillo dll (remember: this dll grows dynamically as each layer is exposed) we need to:

```
/* Determine how much of the actual Armadillo VM is being used for execution
of code */
```

```
Determine_ARMVM(HANDLE thisProcess)
```

This is basically nothing more than iterating through the regions of memory to arrive at the **actual** size:

```
/* cut */
    dwRead = VirtualQueryEx(
        thisProcess,
        dwArmVMNAddress,
        &mbi,
        sizeof(mbi)
    );
    if (dwRead)
    {
        // For this protection, increment Armadillo VM size
        if (mbi.Protect == PAGE_EXECUTE_READWRITE ||
            mbi.Protect == PAGE_EXECUTE_READ)
        {
            // cut */

```

We will then perform:

```
/* Determine if we have IAT elimination */
Determine_Iat_elimination(HANDLE thisProcess)
```

As the functions suggests, we now search for our:

```
// INDEX [1] = IAT ELIMINATION HEX SEARCH STRING
```

in our copied memory block of the virtual armadillo dll address space based on our size determination. Whether or not there is any actual import elimination is not determined by finding this string. It is an integral part of armadillo code. The search string finds the following grey highlighted code offset:

00BBD169	C685 94D4FFFF 01	MOV BYTE PTR SS:[EBP-2B6C],1
00BBD170	8B85 98D4FFFF	MOV EAX,DWORD PTR SS:[EBP-2B68]
00BBD176	40	INC EAX
00BBD177	8985 98D4FFFF	MOV DWORD PTR SS:[EBP-2B68],EAX
00BBD17D	83BD E4D7FFFF 00	CMP DWORD PTR SS:[EBP-281C],0
00BBD184	74 4D	JE SHORT 00BBD1D3
00BBD186	8B85 90D4FFFF	MOV EAX,DWORD PTR SS:[EBP-2B70]
00BBD18C	2B85 E8D7FFFF	SUB EAX,DWORD PTR SS:[EBP-2818]
00BBD192	C1E8 02	SHR EAX,2

Figure 3

Once found, it will calculate the relative displacement for the DWORD PTR variable contained in the instruction for address 00BBD17D [EBP-281C] above.

This is determined by finding our dynamic hex search string “E4D7FFFF” in memory. ArmaGeddon will now find the 1st occurrence of this string and set a software breakpoint on it. We would arrive (see grey highlighting) on this breakpoint:

00BBD127	80A5 94D4FFFF 00	AND BYTE PTR SS:[EBP-2B6C],0
00BBD12E	83BD E4D7FFFF 00	CMP DWORD PTR SS:[EBP-281C],0
00BBD135	75 39	JNZ SHORT 00BBD170
00BBD137	A0 BC15BD00	MOV AL,BYTE PTR DS:[BD15BC]

Figure 4

Note our hex string “E4D7FFFF” above. The CMP instruction compares the DWORD PTR SS:[EBP-281C] with 0. We need to read this pointer address for a value > 0. If the value is > 0, we have import elimination and need to redirect it back to our process’s import section. This is much more preferable to using a tool “after the damage is done” to resolve the problem.

Import redirection, although seemingly simple in concept to execute, is a bit more of a challenge to program.

Ghandi’s excellent tutorial “Unpacking Armadillo 5.xx Protected DLL's (Using Code Splicing protection)_by_Ghandi_v10.pdf” (available on the ARTEAM’s Tutorials section) gave us some hint on how to proceed. This is an excellent source of newer information on the 5.x version you are encouraged to read. Ghandi goes onto to say “The code I am about to show is from Armadillo 5.xx, Armadillo 3.78 – 4.xx is different (yet still easily recognizable once you reverse a few protected files.) And the code for pre-3.78 different yet again.”

Unfortunately, from a programming perspective, this is a significant challenge. Once we have determined IAT elimination, we can navigate (trace into) Import redirection by finding the next “PUSH 100” command. This brings us to our next 2 hex search strings:

```
// INDEX [2] = IAT REDIRECTION HEX SEARCH STRING (PUSH 100)
// INDEX [3] = IAT REDIRECTION HEX SEARCH STRING (CALL ????????, ADD ESP,0C)
```

Once we find the “PUSH 100” we can search for the very next CALL instruction using wildcards. We would see code similar to the following:

00BBD4B9	68 00010000	PUSH 100
00BBD4BE	8D85 58C1FFFF	LEA EAX,DWORD PTR SS:[EBP-3EA8]
00BBD4C4	50	PUSH EAX
00BBD4C5	8B85 58C2FFFF	MOV EAX,DWORD PTR SS:[EBP-3DA8]
00BBD4CB	FF30	PUSH DWORD PTR DS:[EAX]
00BBD4CD	E8 514FFDFF	CALL 00B92423
00BBD4D2	83C4 0C	ADD ESP,0C
00BBD4D5	8D85 58C1FFFF	LEA EAX,DWORD PTR SS:[EBP-3EA8]

Figure 5

The grey highlighted line above is our CALL of interest. To follow this code programatically we need to convert the offset “514FFDFF” to an address we can actually read.

There are a couple ways, I choose to do the following:

Calculation:

0 – FFFD4F51 (reverse order) = DWORD result 0002B0AF

We then subtract the result from our next instruction (address):
 $00BBD4D2 - 0002B0AF = 00B92423$

We can now read address 00B92423 via ReadProcessMemory and write our patch RETN "0xC3" to this address which would look like so in memory:

00B92423	C3	RETN
00B92424	8BEC	MOV EBP,ESP
00B92426	51	PUSH ECX
00B92427	A1 30FFBC00	MOV EAX,DWORD PTR DS:[BCFF30]

Figure 6

This is the only time the program will actually patch any code!
 Afterwards, we do this:

```
// Set a SWBP on CreateThread RETN address
```

This well known technique is used to arrive at the OEP.

1.5.4.3 Copymem-II

For dual process (father + child) targets, we must consider this possibility. There are probably a few different ways to determine. I chose to simply interrogate the protection options in effect for the code section. We can easily do this as follows:

```
VirtualQueryEx(childhProcess, (LPVOID) PESectionAddress, &mbi, sizeof(mbi));
// If PAGE_GUARD protection is in effect then we have copymem2
// This value will be 100 + other protections
if (mbi.Protect > 0x00000100)
{
    // "CopyMem2 detected";
    // Turn on specialized processing
}
```

Otherwise: "No copymem-II"

We set the code section's protection attributes to {PAGE_EXECUTE_READWRITE | PAGE_GUARD}

```
if (!VirtualProtectEx(childhProcess, (LPVOID) PESectionAddress,
    PESectionSize, PEGuardProtect, &PEOldProtect))
{
    ... And so on.
```

Point of note. Mbi represents a pointer to a commonly used structure:

```
MEMORY_BASIC_INFORMATION mbi;
```

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Used in memory specific API's.

In the case of no copymem-II we will simply check the value of Context.Eip for each:

```
case EXCEPTION_GUARD_PAGE:
```

and if the Eip is within the .text (code) section's bounds (memory block), we have reached the proverbial OEP. We may need to keep resetting the protection attributes on the code section until this condition occurs.

For copymem-II, we need to setup a conditional loop. To accomplish our task we:

- save our child DEBUG_EVENT structure.

Remember, the 1st time we reach this exception is our OEP. But now we must decrypt the code section in chunks of 1000 hex byte code blocks the way the armadillo parent would like.

```
// Compute the base exception address
```

- CMeventaddress = our OEP (current page guard exception address)
- CMAaddress = CMeventaddress / 4096 * 4096;

We need this rounded to 1000.

We then need to establish a starting point:

```
// Compute the incremental page guard address
```

- CBAaddress = BaseOfImage; << normally 0x00400000

And we need to establish a Software breakpoint on this OEP (current exception address) for the duration of our copymem-II loop).

- SWBPExceptionAddress[7] = CMeventaddress;

In this case we use index [7] to serve our purpose. This breakpoint will be saved and eventually fire in the child process and be handled by the parent.

We must now perform our final hex string search:

```
// Setup search for computed # .text memory pages (4096 blocks)
// copymem-II for encryption / decryption
```

Armadillo generally doesn't want more than ½ of the available code section decrypted in memory at any time. To prevent this, we need to find the variable containing this limitation and increase it.

For this we need to search the .text1 section of the target process for:

```
// INDEX [0] = COPYMEM2 HEX SEARCH STRING
```

The code we are searching for highlighted below:

00497DFC	3B15 2C984C00	CMP EDX,DWORD PTR DS:[4C982C]
00497E02	0F8E D9010000	JLE D112Lib_.00497FE1
00497E08	50	PUSH EAX
00497E09	F7D0	NOT EAX
00497E0B	0FC8	BSWAP EAX
00497E0D	58	POP EAX

Figure 7

Once we find it we can do the following:

```
// Increase # decrypted pages max variable value in DWORD PTR DS:[4C982C]
```

```
dwBMVMValue = 0xFFFF;
```

and write our DWORD variable value to the found offset address.

Finally, we do this:

```
// Set a SWBP on WriteProcessMemory
```

The purpose of this Software breakpoint (one-time and is handled by the parent process) is to actually set (write) the original page guard exception as a breakpoint exception in the child process.

And now the loop:

Breakpoint exception in child process:

```
// Increment .text section address by hex 1000 bytes
CBAddress += 4096;
if (CBAddress == CAddress)    << CAddress represents original page guard
exception and has already been decrypted!!
    CBAddress += 4096;
// If we have reached the end of the code section
if (CBAddress >= PESectionAddress+PESectionSize)
{
    // display a message: "CopyMem2 completed"; and we are done!
    // now we can go check for the OEP (which we already know)
}
else
```

The parent will not decrypt a breakpoint exception. It expects a page guard exception so we give it one by restoring the original DEBUG_EVENT structure for the child:

```
//Copy saved DEBUG_EVENT structure from Original PAGE_GUARD exception and
update memory addresses for copymem-II in child
memcpy(&CdebugEv, &SebugEv, sizeof(CdebugEv));
```

and update the new addresses in our DEBUG_EVENT structure for incremental page processing (+1000 bytes).

We reset our breakpoint exception and continue our loop until finished!

1.5.4.4 Check for OEP

We've determined that the Context.Eip falls within the code section either via copymem-II or by tracing into the code section in single step mode:

```
/* cut */
// See if we are in the .text section
// If not, reset this event
// Get Thread context
Context.ContextFlags = CONTEXT_FULL;
GetThreadContext(childhThread,&Context);
if (Context.Eip >= PESectionAddress &&
    Context.Eip <= PESectionAddress+PESectionSize)
{
    PEOldProtect = PAGE_EXECUTE_READWRITE;
    if (!VirtualProtectEx(childhProcess, (LPVOID) PESectionAddress,
        PESectionSize, PEOldProtect, &PESecProtect))
    {
        sprintf( b,"Unable to Remove GUARD_PAGE protect\n"
            "Exception Address: %08X", PESectionAddress);
        MessageBoxError(b);
        contproc = FALSE;
    }
}
else
{
    // Restore original OEP byte for copymem2
    if (scanbyte[7] != 0x00)
    {
        WriteProcessMemory (childhProcess, (LPVOID)
            SWBPEExceptionAddress[7], scanbyte[7],sizeof(BYTE),
            &dwWritten);
    }
}
```

```

    }
    OEPEAddress = (DWORD)
    CebugEv.u.Exception.ExceptionRecord.ExceptionAddress -
    BaseOfImage;
/* cut */

```

If the EIP is not within the code section, repeat steps (trace) until TRUE.

What is important is to insure our EIP is within the code section rather than reference a read / write operation performed against it!! Then:

- Restore protection on the code section.
- Restore original byte before breakpoint exception.

At this point, the target process is at a dead stop, all threads are suspended and is ready to dump and rebuild. One important note: The process is still continuable. This is necessary for analyzing / logging nanomites.

Should the automatic Import Reconstructor fail to find / fix imports, a workaround would be to “Cancel” the dump option and instead use LordPE or similar tool to dump the target application and use ImpREC to fix the imports.

... Let the conflict begin!

The End

1.6. References

- Z0mbie - <http://mirror.sweon.net/z0mbie/xde102.zip> - download z0mbie's extended length disassembler engine source
- Nacho_dj - ARTeam Import Reconstructor 1.0 (ARImpRec.dll)
- Admiral - Arminline v0.96f source code (nanolib.dll) <http://arminline.byethost22.com/>
- DLL2LIB - <http://www.binary-soft.com> homepage
- AndreaGeddon [Reversing Armadillo 4.20](#) Tutorial - http://www.quequero.org/Reversing_Armadillo_4.20
- Ghandi - Tutorial: Unpacking Armadillo 5.xx Protected DLL's (Using Code Splicing Protection) <http://arteam.accessroot.com/tutorials.html?fid=205>
- www.tuts4you.com Tutorials - <http://www.tuts4you.com/download.php?list.11> + whole lot more
- Arteam Tutorials and a whole lot more - <http://arteam.accessroot.com/tutorials.html>

1.7. Conclusions

I had hoped that this would be a relatively short overview, but there is a considerable amount of information which goes into the making of a product and then explaining its internals. I hope you enjoyed reading it. I hope you understood most of it. More importantly, I hope you, the reader, may be encouraged to contribute something worthwhile to the RCE community and write a Tutorial or Ezine about it. I would like to point out that the Armageddon tool was a collaborative venture and would not be possible were it not for the efforts and contributions of people devoted to RCE.

1.8. Greetings

A hearty “Greetz” go out to:

SSIEvIN for the GFX work

Nacho_dj, deroko, MaDMAN_H3rCuL3s for testing

Lena151 for the IARP protection system

[ARTEAM], [SnD], [ICU], [RES], [RET], [CracksLatinoS]

The RCE forums and all the people who have contributed to the scene :-)

Document History

- Version 1.0 first public release

