CONDZERO - ARTEAM

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

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  .aabbc   3000
  0044B000 - 00479000  .ddff    2E000
  00479000 - 004C9000  .gghhi   50000
  004C9000 - 004D9000  .jjkkl   10000
  004D9000 - 004F9000  .mmnnp   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"

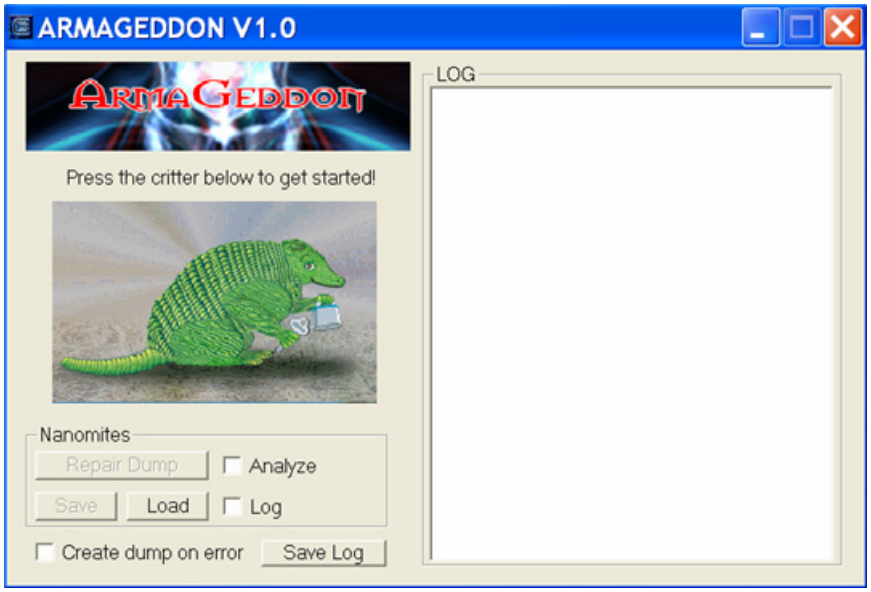

Figure 1

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 Z0MBiE'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,
```