Development & Security

By Jurriaan Bremer @skier_t

Malware Unpacking Level: Pintool

Posted on **June 18, 2012**

Table of Contents:

Abstract

Today’s post presents an introduction to Pin [1], as well as a tool which can automatically unpack so-called RunPE malware, malware that has been *packed* (in a specific way) in order to avoid detection by Anti-Virus software.

Throughout the following paragraphs we will introduce the reader to Pin and present the reader some example uses. Then we will get to RunPE malware, how it works, and how we will unpack it.

Finally, we present the reader source and binaries of the Pintool to unpack RunPE malware, as well as a usage guide.

Note: When running the tool presented in the Proof of Concept section with real malware, use a Virtual Machine! I’m not responsible for any damage.

Introduction

A brief introduction on Pin from the official Pin website.

```
Pin is a dynamic binary instrumentation framework for the IA-32 and x86-
64 instruction-set architectures that enables the creation of dynamic
program analysis tools. ... The tools created using Pin, called Pintools,
can be used to perform program analysis on user space applications in
Linux and Windows. As a dynamic binary instrumentation tool,
instrumentation is performed at run time on the compiled binary files.
Thus, it requires no recompiling of source code and can support
instrumenting programs that dynamically generate code.
```

In other words, a Pintool allows **complete control** over runtime execution. It is, for example, possible to catch all *call* instructions (e.g. see the following post [2]), so you can determine which functions are being called by a program.

Pin offers a rich API to log, alter, or do anything else with an existing compiled binary. For example, it is especially useful in situations where one might want to analyze a program, but doesn't have the source code of said program.

For another excellent introduction to Pintools, with topics such as taint analysis and automated exploit generation, please refer to the work by Gal Diskin [3].

Otherwise, simply remember that we have full control over the execution of a binary, which means we can alter execution flow as we like.

RunPE Introduction

RunPE is a method originally invented by Tan Chew Keong [4] (correct me if I'm wrong) as possible way to *fork* [5] a running windows application. Since a few years it has been used to successfully bypass software such as Anti-Virus'. There have been a handful of approaches to automatically *unpack* this kind of malware, that is, extract the original *payload*. (The real malware, the payload, is usually embedded in a packer and executed runtime.)

However, previous work used hardware breakpoints [6] or attached a debugger [7] to set software breakpoints. These approaches are vulnerable to those RunPE binaries which do system calls directly. For example, it is common for RunPE malware to call the normal *WriteProcessMemory* [8] API, which writes data to another process. However, malware can go deeper by calling *NtWriteVirtualMemory* [9] directly. If malware does so, any RunPE unpacker which sets a breakpoint on WriteProcessMemory will not be able to unpack this malware (internally WriteProcessMemory calls NtWriteVirtualMemory, so setting a breakpoint on a higher level API will not catch calls to the lower level function.)

Now this should still be fine (one could simply set a breakpoint on NtWriteVirtualMemory instead of WriteProcessMemory), but what if malware executes system calls directly, instead of calling NtWriteVirtualMemory? For example, the machine code located at the address of NtWriteVirtualMemory could be copied into another location, and from there, the malware could, instead of calling NtWriteVirtualMemory, call the other location, which results in the same system call, but the unpacker would not be able to detect this.

This is where Pin comes into play. A Pintool has the ability to trace *every* instruction of a process in almost realtime (depending on your pintool a performance decrease of a few times is likely, which is really fast.) Since Pin gives us the opportunity to analyze every instruction that will be executed, we can give some extra attention to system calls.

Intercepting System Calls using Pin

In a previous post we discussed interception of system calls under wow64. Pin provides similar functionality, with the added advantage that it works for architectures without wow64 as well (that is, 32bit and 64bit windows with 64bit binaries.) Besides that, a pintool is also better performance-wise.

Enough about why Pin is so awesome, let's get to a simple example which we will be extending later on. Pin provides two functions which give us full control over system calls, namely *PIN_AddSyscallEntryFunction* and *PIN_AddSyscallExitFunction* [10]. As the names suggest, these functions provide us the ability to add extra functionality whenever a system call occurs. For example, when a process is ran under our pintool, and we have installed callback functions for both syscall entry and exit, then our callback functions will be called before performing a system call *and* after the system call.

The following code snippet represents a simple pintool which dumps the system call number, the first four parameters to the system call, as well as the return value of every system call that's being performed.

```
#include <stdio.h>
#include "pin.H"

void syscall_entry(THREADID thread_id, CONTEXT *ctx,
    SYSCALL_STANDARD std, void *v)
{
    printf("system-call: %d, arguments:",
        PIN_GetSyscallNumber(ctx, std));
    for (int i = 0; i < 4; i++) {
        ADDRINT value = PIN_GetSyscallArgument(ctx, std, i);
        printf("  %d 0x%08x", value, value);
    }
}

void syscall_exit(THREADID thread_id, CONTEXT *ctx,
    SYSCALL_STANDARD std, void *v)
{
    ADDRINT return_value = PIN_GetSyscallReturn(ctx, std);
    printf(", return-value: %d 0x%08x\n", return_value,
        return_value);
}

int main(int argc, char *argv[])
{
    if(PIN_Init(argc, argv)) {
        printf("Usage: %s <binary> [arguments]\n");
        return 0;
    }

    PIN_AddSyscallEntryFunction(&syscall_entry, NULL);
    PIN_AddSyscallExitFunction(&syscall_exit, NULL);

    PIN_StartProgram();
    return 0;
}
```

As you can see, creating a pintool is really easy. The output when running a program under this pintool is also fairly straightforward, it simply prints all the system call numbers with parameters and return values.

It is interesting to note that *SYSCALL_STANDARD* [11] is set to a different value depending on the Operating System and architecture (and therefore calling convention.) Other functions provided by the Pin API, such as *PIN_GetSyscallArgument*, depend on this value, because based on this value they know whether the arguments to a system call are stored in registers or on the stack etc (this varies per calling convention.)

RunPE Unpacking

For a detailed reference to the internal workings of RunPE feel free to read either the original description by Tan Chew Keong [4], or the code that can be found as *poc2.c* in the Proof of Concept section.

The method of RunPE only contains three interesting steps for us:

1. a new process is created, with the main thread suspended
2. memory in the suspended process is overwritten with a payload
3. the main thread is resumed

Now, before we go any further, each of these operations is done through a system call. For example, overwriting data in another process is done using WriteProcessMemory, which in turn calls NtWriteVirtualMemory, which results in a system call.

With the information that each of these operations is performed using a system call, we can be sure that Pin will catch and intercept the system calls for us, after which we can process them. Without further ado, unpacking RunPE binaries with the three steps listed above.

The first step, creating a new process, is where we start monitoring for overwriting memory in the newly created process.

In the second step the RunPE packer will overwrite the memory contents in the other process, we will intercept and dump all data that is used to overwrite the memory in the other process.

Finally, in the third step, the main thread is resumed. Which indicates that all data has been overwritten in the other process. Because when the main thread is resumed, the windows loader will kick in and execute the executable that is loaded into memory. By killing both processes at this time we disallow any malicious code to run (the payload that was written to the other process should be considered malicious, whereas the RunPE packer itself is usually fairly harmless.)

The three steps above show a simplified version to unpack most (if not all) of the RunPE *crypters* in the wild (crypter, packer.. it’s all the same.) Luckily for us, it won’t get much harder than this.

So, basically, after running our tool we have a dump containing the address, length of the block and a raw dump of the block for each WriteProcessMemory call (or lower variants, our Pintool catches the lowest variant, so it doesn’t really matter) that occured. The Proof of Concept section also provides a simple python script which transforms the raw dump into a valid PE (windows binary) file.

RunPE Improvements

Although unseen on RunPE malware in the wild, the following techniques could be used to make it a little harder for our unpacker.

- create multiple processes, only overwrite one with real data
- duplicate process handles
- open and close handles to the process
- use GhostWriting technique

By creating multiple processes, our unpacker tool should now keep track which data is written to which process. By duplicating handles (obtain a new handle with the same purpose as the original handle) we could work further on this, although both of these techniques could be easily adapted by our unpacker.

If the RunPE packer opens new handle to a process it has just created, then that will take a little extra effort as well, as our unpacker would have to link the process identifiers. (From there it would be the same as duplicating handles.)

And finally, one could use the GhostWriting technique, which (although I was unaware of this term at the time of writing) can be found in a previous post.

Proof of Concept

Up-to-date source of the Proof of Concept can be found here. Binaries (with source as well) can be found here.

Instructions to build your own pintool are as follows, use them, or it will break everything (trust me..)

Install Pin [12] to your favourite directory, create a new directory called **godware** in $PIN/source/tools and unzip *runpe-pin.tar* to this directory. (E.g. $PIN/source/tools/godware/godware.cpp should point to godware.cpp)

Now open the Visual Studio Command Prompt (yes, it’s ugly), go to the

$PIN/source/tools/godware/ directory, and run the following command: **..\nmake**. This will successfully build the godware.dll component.

You can compile *poc2.exe* using gcc/mingw (see the Makefile for commandline arguments.) poc2.exe does the following, it loads a file specified by the first commandline argument into memory and after that, uses the RunPE method to execute that file (i.e. in another process.)

Note: Yes, there is a Nmakefile and a Makefile in the tar file, the Nmakefile is used by the windows nmake utility, whereas the Makefile is for the Cygwin make utility.

The following illustrates the dumping of the RunPE method and the rebuilding of the PE file.

```
# make sure you are in the $PIN/source/tools/godware directory
$ ../../../ia32/bin/pin -t obj-ia32/godware.dll -- poc2.exe msgbox
... snip: a lot of dumped system calls ...
$ python rebuild.py logz.txt
... snip some debug data ...
$ mv logz.txt.out output.exe

# execute the restored binary, don't do this on malware..
$ output.exe
```

Hopefully that was clear enough, because Pin is kind of messy to build.

The Proof of Concept has been tested successfully on poc2.exe, Wind of Crypt, and p0ke Crypter. For more malware samples, check out malware.lu!

That was it for today, if any questions arise, you know where to find me 😉

References

1. Pintool – Intel
2. Solving Nuit Du Hack Challenge using Pin
3. Gal Diskin's Hack in the Box 2012 Amsterdam Materials
4. LoadEXE: Dynamic Forking of Win32 EXE – Tan Chew Keong
5. fork(2) – linux
6. Unpacking RunPE using Hardware Breakpoints – Interesting Malware
7. Unpacking RunPE by attaching a debugger – Thunked
8. WriteProcessMemory – MSDN
9. NtWriteVirtualMemory – NT Internals
10. Pin System Call API – Intel
11. SYSCALL_STANDARD – Intel
12. Download Pintool – Intel
13. make – linux

This entry was posted in **Uncategorized** by **jbremer**. Bookmark the **permalink [http://jbremer.org/malware-unpacking-level-pintool/]** .

8 THOUGHTS ON “MALWARE UNPACKING LEVEL: PINTOOL”

Nick
on **October 24, 2013 at 10:36 am** said:

Hi my dear,
thanks for the nice post. Nmake is abandoned since a certain while so I tried to compile it with VS 2010. Unfortunately it cannot find the functions GetLock, ReleaseLock and InitLock (Locking Primitives)
1>MyPinTool.cpp(248): error C3861: “GetLock”: identifier not found
1>MyPinTool.cpp(259): error C3861: “ReleaseLock”: identifier not found
1>MyPinTool.cpp(318): error C3861: “InitLock”: identifier not found

Any ideas?
Cheers, Nick

> computerline
> on **April 8, 2014 at 4:47 am** said:
>
> you can use PIN_GetLock, PIN_ReleaseLock, PIN_InitLock instead

Nick
on **October 25, 2013 at 6:47 am** said:

Never mind, found the “problem”……..

> jbremer
> on **November 3, 2013 at 9:43 am** said:
>
> Would be nice if you could provide the solution as well I’m sure you’re not the first, or the last, to encounter this problem
>
> > **alex**
> > on **November 13, 2013 at 1:39 am** said:

Hello. I had the nmake issue earlier. It can be solved by installing make via cygwin. During the cygwin install, under Select Packages select Devel > make: The GNU version of the ‘make’ utility. Once installed add the cygwin\bin path to the environmental variables. After that open up the visual studio command line and it should compile. I didn’t try it with your code but the samples in the pin source tools dir. Hopefully that will help others. Cheers.

Nick
on **December 10, 2013 at 11:39 am** said:

The problem was that I used the most recent version of PIN (2.13). Using version 2.12 works flawlessly.
Cheers, Nick

Pingback: Dynamically Unpacking Malware With Pin | My reading

Carlos
on **July 12, 2014 at 5:52 pm** said:

Hi, this may come a bit late 😉 but the problem is that in 2.13 all locking primitives were renamed with a “PIN_” prefix. I hate it when they do that… 😛