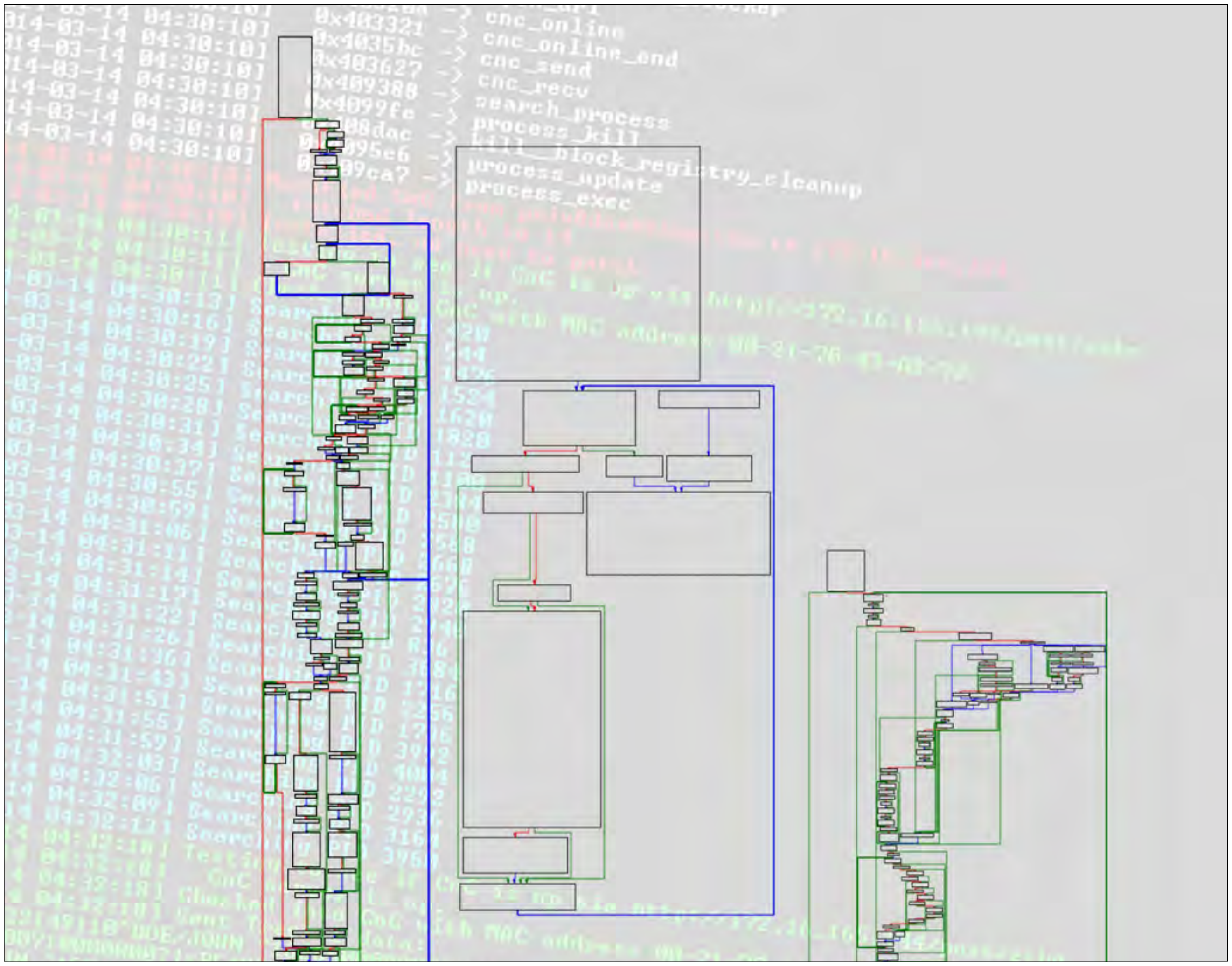


Instrumenting Point-of-Sale Malware

A Case Study in Communicating Malware Analysis More Effectively

Robert Wesley McGrew, Ph.D.
Assistant Research Professor
Department of Computer Science and Engineering
Distributed Analytics & Security Institute (DASI)
Mississippi State University

July 13, 2014

ABSTRACT

The purpose of this white paper, and the talk that accompanies it, is to promote the adoption of better practices in the publication and demonstration of malicious software (malware) analyses. For various reasons, many popular analyses of malware do not contain enough information required for a peer to replicate the research and verify results. This hurts analysts that wish to continue to work more in-depth on a sample, and reduces the value of such analyses to those who would otherwise be able to use them to learn reverse engineering and improve themselves personally. This paper and talk proposes that we borrow the concept of "executable research" by supplementing our written analysis with material designed to illustrate our analysis using the malware itself. Taking a step beyond traditional sandboxes to implement bespoke virtual environments and scripted instrumentation with commentary can supplement written reports in a way that makes the analysis of malware more sound and useful to others.

As a case-study of this concept, an analysis of the recent high-profile point-of-sale malware, JackPOS is presented with enough information to replicate the analysis on the provided sample. The command-and-control server is included and Python-based harnesses are developed and presented that illustrate points of interest from the analysis by instrumenting the execution of the malware itself.

INTRODUCTION

This paper serves as a companion to a talk/demonstration given by the author on improving published malware analyses in general, and resources provided (code and configuration) that allow the reader to follow along with, reproduce, verify, and resume work from the case study analysis. We will begin with a discussion on the reproducibility and verifiability of published malware analyses, then make recommendations for improving these traits in future analyses. Finally, a case study using a recent and popular point-of-sale malware sample is presented to demonstrate how these recommendations can be put to practical use.

REPRODUCIBILITY AND VERIFIABILITY IN MALWARE ANALYSIS

The author, in the course of learning about malicious software (malware) analysis, conducting research in malware analysis, and accumulating materials suitable for teaching reverse engineering (with a focus on malware), has read many malware analyses published by companies, organizations, and individuals. While many consumers of such analyses never intend to "get their hands dirty" and verify results or use them as a basis for further work, and are perfectly satisfied with taking the word of the analyst. This is fine for those focused on indicators of compromise and removal instructions, as well as those who only need an "executive summary" of operation. Those involved in malware research that are interested in the low-level operation of malware, in the pursuit of such things as attribution, documentation of techniques, and comparison of samples, are often left with less information than they'd like to begin their work. Students learning reverse engineering in the context of malware analysis are even poorer served, without documentation of how to reproduce the findings of an analysis, an activity that would otherwise be very educational.

Replicating the results of an experiment as a part of peer review is a cornerstone of the scientific method[1]. Many published analyses of malware have critical elements missing that prevent the analysis from being as useful as it could be to other analysts:

Sample information - By far, the element necessary to replicate results that is most frequently missing is availability of the malware sample itself. We will discuss the reasoning behind this later in this paper, but often, a published analysis references a malware sample that is unavailable for other researchers to download. Less frequently, the specific identifying hash of the malware sample is not specified, or a body of hashes is provided, only one of which (unspecified) is the variant that is the subject of the analysis.

Procedural information - This is in reference to procedures used to analyze the malware. While it's not expected that every step be documented (general procedures are well documented for beginners[2]), those specific to the malware itself that might stand in the way of an analyst are often not well documented. As an example, procedures used to extract packed samples to a state suitable for analysis are often not well-documented.

Contextual information - Most frequently missing, in this category, is information about infection sources and command and control hosts. Often this is related to missing sample information in that the very information that could be used to acquire samples discussed in analyses is often redacted in screenshots, log files, and other documentation of where the malware originated, is hosted, or "phones home".

Internal points of reference - While many analyses discuss functionality and algorithms internal to the malware, and sometimes a block diagram of the basic execution of the sample, information about where those elements are located within the sample itself is often omitted.

Not all analyses are missing all of the above information, and, with effort, some missing elements can be reconstructed, though when this the case, it represents what could be made an unnecessary duplication of effort.

RATIONALE FOR LACK OF REPRODUCIBILITY

The author acknowledges that the “problems” discussed in the previous section are often intentional on the part of those that publish analyses. The absence of specific information is not necessarily an indicator of poor quality of analysis or analyst. The following are presumed rationale for excluding information that would be necessary for reproducibility and verifiability. These points are considerations that must be made before publishing more detailed analyses, and therefore represent hurdles to the adoption of the author’s proposed improvements to published analyses.

Target Audience - In many cases, the target audience does not include fellow analysts or students. If the analysis is intended to deliver indicators of compromise and removal instructions to those who are tasked with defending networks without requiring detailed knowledge of malware internals, then the bar for reproducibility is much lower.

Added Effort - Designing an analysis report in a way that includes information needed for reproducibility certainly requires more effort than a report that omits such information. This is compounded when detailed notes are not kept during the analysis with the future intention of putting together a more “complete” report.

Analysis Consumer Safety - Information may be redacted to *prevent* readers from acquiring, running, and self-infecting with malware in an uncontrolled way. Some organizations may see this as a liability issue.

Client Confidentiality - Information may be withheld that is specific to the client of the reporting analyst, due to confidentiality agreements. Information may be redacted in an over-zealous manner, or the malware sample itself might be withheld due to not being able to say with certainty what client data might exist in the sample. The author of this work sees this as being the most compelling argument *against* including extended information in an analysis.

Competitive Advantage - For many companies and organizations, a popular published analysis is a public relations boon, showcasing technical capabilities and providing a venue for advertising products and services. By publishing an analysis, and withholding information that could be used to replicate that analysis, analysts at competing organizations are unable to effectively leverage the provided information to produce "better" analyses. In the case of malware samples that are not widely available, a company with exclusive access to a particular sample has the advantage of being the only source of information about that malware.

While all of the above may be valid rationale under some circumstances, it is worth pointing out that a decision to redact or withhold information for *competitive advantage* can almost always be outwardly claimed to be due to any of the other reasons. While the author is not aware of any hard evidence that competitive advantage is the rationale behind many analyses falling short of reproducibility and verifiability, it has been anecdotally observed that independent, individual, and academic analysts' analyses are, on the whole, more likely to contain information supporting reproducibility than those published by larger companies (especially those in the antivirus industry).

BORROWING FROM "EXECUTABLE RESEARCH PAPERS"

In academia, scholarly works are subject to peer review that, when the system of peer review works, can allow others to "stand on the shoulders of giants" to perform their own work, as well as expose flaws in research when published results are unable to be verified by other researchers. In computational science, a field that involves using computing and data analysis to solve problems[3], there has been a movement towards developing and using a system of "executable research papers", where research can be presented in electronic and interactive form that embody the algorithms, data, and analysis of results in a way that the reader can execute, observe, and manipulate[4].

While it may not be prudent to create malware analysis reports that are immediately executable, it can be argued that providing samples, resources, and instrumentation for demonstrating findings alongside an analysis brings the concept of published analysis closer to that of an “executable paper”. A system similar to that proposed by the SHARE project (as a response to the Executable Paper Grand Challenge) [5], where research is presented in the context of a virtual machine containing the software and data required to replicate it may have more immediate application to publishing “high information content” malware analyses.

RECOMMENDATIONS

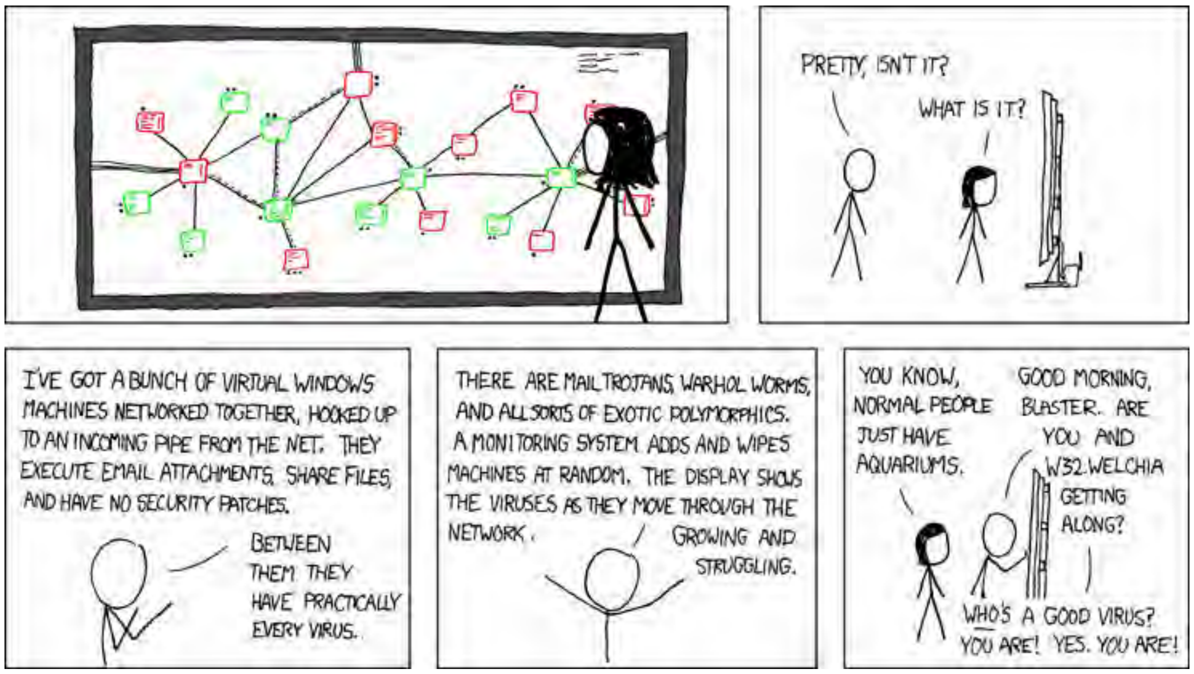
It is proposed that, for a published analysis of malware to be considered easily reproducible and verifiable, the following traits may be desirable:

- **Sample Availability** - Where the medium used to communicate the analysis does not lend itself to safely distributing a live malware sample, the sample should be made available to download from a site such as VirusShare[1], which has been set up as a central repository for making samples easily accessible by researchers.
- **Host Environment** - Beyond what is provided by traditional “triage” tools for malware analysis sandbox environments, an environment should be provided, or documented in enough detail for an analyst to reproduce, that allows for the primary functionality of the malware to execute. Operating system and targeted software should be documented.
- **Target Data -** When malware seeks data to exfiltrate, software to generate and simulate that data can be provided to exercise those areas of malicious code.
- **Network Environment** - Network resources needed to exercise the primary functionality of the malware should be provided or documented. For example, malware that communicates with a command and control server should be accompanied by an implementation of the command and control protocol.

Developing and providing a command and control server, where one is not already widely available, may, in some cases, involve large investment of time. In the author's reverse engineering course, students create command and control servers for undocumented samples. A minimal implementation, or, at least, as complete documentation of the protocol as needed to demonstrate functionality should be provided.

- **Instrumentation** - By use of scriptable debugging, such as that provided by the Python library WinAppDbg [10], a "harness" can be created that executes the malware in a way that is easy to observe, and illustrates the findings of the analyst's static and dynamic analysis. As the analysis process progresses, each finding regarding internal data or functionality can be illustrated at a programmatic breakpoint in the actual malware code. Essentially, debug output for the malware can be created where there originally was none. Harness scripts can also be created that execute isolated sections of the malware to illustrate the operation of functions deeper in the path of execution independently of the rest.

Taken as a whole, the written analysis, combined with the above electronic deliverables, can be used to create en environment in which the malware can be observed at a high or low level, and manipulated to further that analysis. For those who enjoy and engage in malware reverse engineering, it's a step closer to owning and maintaining the "Malware Aquarium" as described in XKCD 350 [6]:


PRETTY, ISN'T IT?
WHAT IS IT?
I'VE GOT A BUNCH OF VIRTUAL WINDOWS MACHINES NETWORKED TOGETHER, HOOKED UP TO AN INCOMING PIPE FROM THE NET. THEY EXECUTE EMAIL ATTACHMENTS, SHARE FILES, AND HAVE NO SECURITY PATCHES.
BETWEEN THEM THEY HAVE PRACTICALLY EVERY VIRUS.
THERE ARE MAILTROJANS, WARHOL WORMS, AND ALL SORTS OF EXOTIC POLYMORPHICS. A MONITORING SYSTEM ADDS AND WIPES MACHINES AT RANDOM. THE DISPLAY SHOWS THE VIRUSES AS THEY MOVE THROUGH THE NETWORK,
GROWING AND STRUGGLING.
YOU KNOW, NORMAL PEOPLE JUST HAVE AQUARIUMS.
GOOD MORNING, BLASTER. ARE YOU AND W32.WELCHIA GETTING ALONG?
WHO'S A GOOD VIRUS? YOU ARE! YES. YOU ARE!

CASE STUDY: JACKPOS POINT-OF-SALE MALWARE

The purpose of this section and its corresponding demo that makes up a good portion of the accompanying talk is to illustrate how an analysis can be published that discusses the operation of malware at a high level, while also providing the detail needed for analysts to run the malware in a safe environment and replicate its results.

Sample Selection

The JackPOS sample was chosen largely by virtue of being the most current piece of malware being analyzed by the author of this work at the time this paper was created. During the time the author was performing his analysis and creating this work, other analyses and writings on this malware have been published. Notably, Josh Grunzweig of TrustWave Spiderlabs has published a blog post discussing a brief analysis of JackPOS [7], and unixfreakjp of the Malware Must Die collective has discussed the sale and distribution of this malware[8]. It is the author's opinion that Grunzweig's analysis is quite good, and contains much of the information needed to reproduce the analysis, even if all of the recommended elements of a reproducible and verifiable analysis are not present.

A very special thanks from this author goes to Xylit0l of Malware Intelligence for running the CyberCrime Tracker, which allowed the author to locate a collection of JackPOS samples and even capture a copy of the command and control server.

The general consensus among all of the current analyses of JackPOS is that it is not a terribly impressive piece of malware. There are a few reasons, though, to justify it as a good choice for a case study of this kind:

- Point-of-Sale malware is a currently growing concern, so illustrating an example of it makes for a compelling demo.
- A copy of the malware operator's command and control source code has been acquired (and distributed along with this analysis), which gives us the opportunity to watch execution from both the client and server's perspective. We can demonstrate a full environment for operating, and being infected by, the JackPOS malware.
- The sample's use of C++ strings and STL constructs makes casual static analysis of the disassembly more challenging than it would if it were written in more straightforward C. The superficial use of objects created at runtime confuses stock IDA Pro's ability to cross reference usage of data. This makes the test harnesses provided more useful in documenting the state of important memory locations at points of interest at runtime.
- The functionality for searching other running processes can be instrumented to run independently of the remainder of the malware.
- It's relatively easy to patch in a way that runs in the harness without its normal installation and persistence setup.

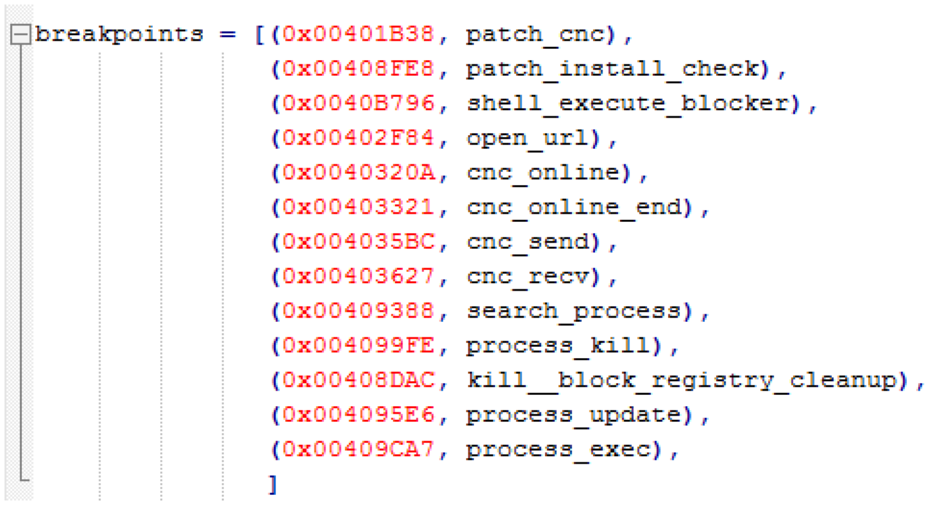
What's Provided

- The sample itself is available in VirusShare.com archive, to avoid providing live malware with conference materials.
    - MD5 - aa9686c3161242ba61b779aa325e9d24
    - SHA1 - 9fa9364add245ce873552aced7b4a757dceceb9e
- PHP source code for the command and control server
- A database schema created to work with the above command and control
- `jackpos_harness.py` - A Python harness for instrumenting the execution of the JackPOS sample. Used to patch JackPOS at runtime to connect to an arbitrary command and control server, execute it in a way that's friendlier to analysts than its usual installed and persistence state, and illustrate the internal operation of JackPOS as debug output as it executes.
- `search_proc_harness.py` - Another harness script that bypasses the majority of JackPOS functionality in order to *only* call the process memory searching functionality on a PID given at the command-line.
- gen_track_data.py - A script that acts as a (generated) data source for JackPOS to gather from and exfiltrate. Essentially, this is a stand-in for the point of sale terminal software.

With a combination of the analysis and the resources provided, it is possible to set up an environment that executes and demonstrates the entirety of the "desired" path of execution and data (from card "swipe" to showing up in the command and control) of JackPOS. This virtual environment will be presented and discussed as part of the accompanying talk to this paper.

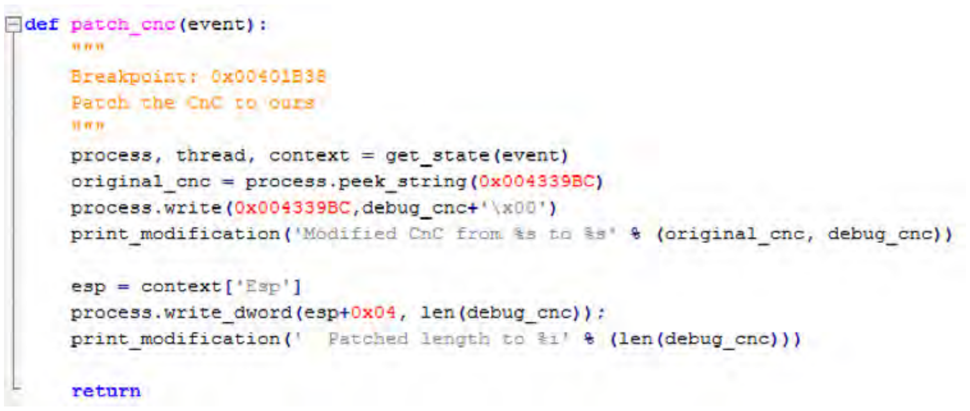
Harness Design

The two Python harnesses provided are both based on a very simple template that is intended to be easily applied to the analysis of other samples. The WinAppDbg[10] library is used to implement scriptable debugging of the target malware sample. While the operation of these harnesses can be flexible, most features are implemented by functions that trigger upon hitting breakpoints set in code. A list of breakpoints with associated callbacks exists in the code:



```
breakpoints = [(0x00401B38, patch_cnc),
               (0x00408FE8, patch_install_check),
               (0x0040B796, shell_execute_blocker),
               (0x00402F84, open_url),
               (0x0040320A, cnc_online),
               (0x00403321, cnc_online_end),
               (0x004035BC, cnc_send),
               (0x00403627, cnc_recv),
               (0x00409388, search_process),
               (0x004099FE, process_kill),
               (0x00408DAC, kill__block_registry_cleanup),
               (0x004095E6, process_update),
               (0x00409CA7, process_exec),
               ]
```

Callbacks receive an event, then typically extract process, thread, and context information from the break event. The purpose of most of the callbacks is to illustrate points of interest in the malware by displaying the status of important areas of memory, indicating that execution has reached a certain point, or patching the target malware in order to reach desired code. The following code that patches the command and control server string at the appropriate point in execution demonstrates how the callbacks work:



```
def patch_cnc(event):
    """
    Breakpoint: 0x00401B38
    Patch the CnC to ours
    """
    process, thread, context = get_state(event)
    original_cnc = process.peek_string(0x004339BC)
    process.write(0x004339BC,debug_cnc+'\x00')
    print_modification('Modified CnC from %s to %s' % (original_cnc, debug_cnc))

    esp = context['Esp']
    process.write_dword(esp+0x04, len(debug_cnc));
    print_modification('  Patched length to %i' % (len(debug_cnc)))

    return
```

For debug output, several wrapper functions around WinAppDbg's color output functions are provided to color-code output in order to make identifying different types of output easier, and apply timestamps to each line. For the case study analysis the following conventions are used for color-coded output:

Output Function	Color	Output Type
`print_script(str)`	**White**	Information from the harness itself
`print_modification(str)`	**Red**	Modifications made at runtime
`print_hook(str)`	**Yellow**	Hooked API functions
`print_network(str)`	**Green**	Network communication
`print_process(str)`	**Cyan**	Information regarding malware querying processes
`print_internal(str)`	**Magenta**	Internal operation/data in the malware

ANALYSIS

Overview

JackPOS is a simple implementation of point-of-sale malware. The specific sample studied in this analysis matches the SHA1 sum:

```
9fa9364add245ce873552aced7b4a757dceceb9e
```

The malware supports installing itself, setting up registry persistence, and executing a "watchdog" process that will re-spawn the malware if it is killed or crashes. The main body of the malware's functionality involves a loop in which the malware scans processes on the system, and uses a pair of search functions on regions of target process memory to look for "dumps" of credit card magnetic stripe track data. The malware then sends the track data to the command and control server, and at that time has the ability to receive commands from the server to update itself, upload and execute another program, or kill its own process.

The focus of this analysis is on what is considered to be primary functionality: searching for credit card track data and communications with the command and control.

Command and Control

The command and control server for JackPOS, captured from a live site, takes the form of a PHP web application that uses the Yii framework[11]. The PHP code has been included with the distributable material for this talk as `jackpos_cnc.tar.gz`. Some configuration files have been modified from the captured version in order to run the server in the analysis environment. For the analysis environment, a fresh Linux Mint 16 was created to act as the command and control server, with `tasksel` installed in order to get a base `lamp-server` setup. In addition to the base LAMP stack, three other packages should be `apt-get install`'d:

- `php5-imagick`
- `php5-gd`

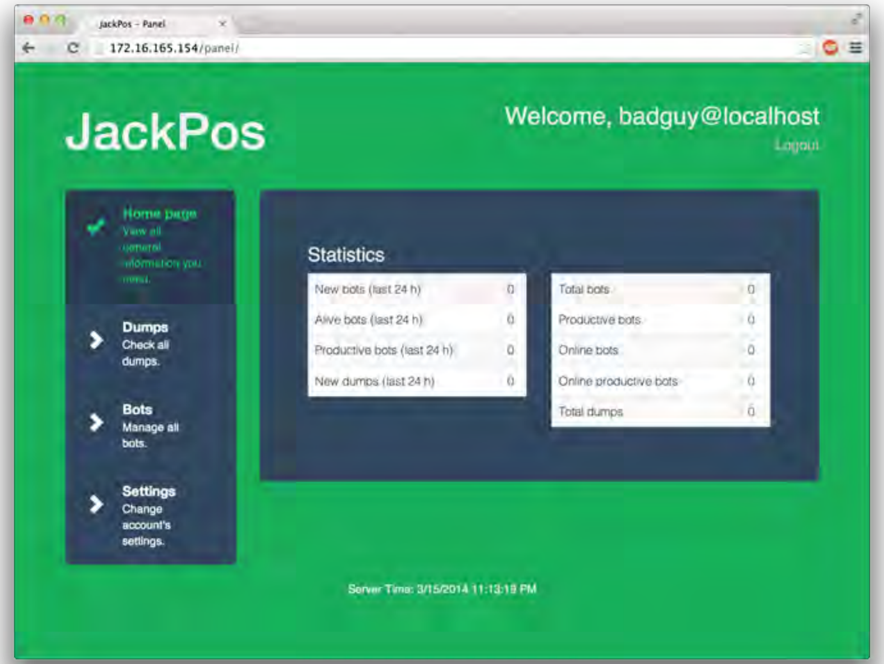
- `php5-json` (without which, frustratingly, select portions of the interface are simply blank with no error)

Regrettably, the captured command-and-control server did not include a schema for the database, and unlike other MVC frameworks like Django, Yii projects do not support creating a database schema from the data model definitions. A schema has been created through a combined process of addressing errors regarding missing tables and columns as they arise, and determining needs by inspecting the data model source code in the `protected/models` directory. This crafted schema has been included in the file, `jackposc2_schema.sql`, which can be imported into a freshly-created database named "jackposc2" in order to provide the command and control with the database it's expecting. The data model and schema creation process illustrates the primary focus of this malware and its command and control, with tables that include:

- `bots` - Identified by MAC and IP addresses
- `cards` - With related "`dumps`" and "`tracks`" tables
- `commands` - To queue commands to be sent to hosts when they check in

In `jackposc2_schema.sql`, there is also a single user record installed, so that the analyst can log into the interface, navigate around, and see when the malware checks in and submits data. To log in, use email address "badguy@localhost" with the password "jackpos".

Once set up successfully, the command and control server should be up and running for the analyst to log into and see the following interface:



Statistics

New bots (last 24 h)	0
Alive bots (last 24 h)	0
Productive bots (last 24 h)	0
New dumps (last 24 h)	0

Total bots	0
Productive bots	0
Online bots	0
Online productive bots	0
Total dumps	0

Server Time: 3/15/2014 11:13:19 PM

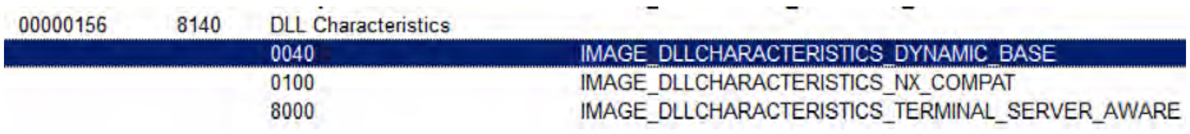
The “Dumps”, “Bots”, and “Settings” pages should be operational, though there is not yet any data from infected clients to view or interact with. We’ll revisit this interface after we have set up the sample to communicate with it.

First Steps with the JackPOS Client Sample

The sample is packed using UPX, as can be noted by examining it with PEID or RDG Packer Detector. This is fortunate, as it allows us to avoid a lengthy discussion of unpacking and move on to the analysis relatively quickly. The sample was unpacked using the command line UPX packer with the “-d” option.

While the sample was apparently successfully unpacked, judging from the immediate presence of program-unique strings and code that uses them in static analysis, there was a problem with the unpacking process. Attempts to run the unpacked program caused it to immediately crash. Immunity Debugger reports the crash as a result of an “Access violation when reading [00438818]”. According to IDA Pro, this *would* be the address of a stack cookie seed value, but only *if* the executable loaded at it’s “preferred” base address of 0x00400000. Due to ASLR, it’s not loaded at that address, and the relocation information needed to fix this reference (and potentially others) is missing.

A fix for this is, or at least one that is suitable for malware analysis, is simple: disable relocation for this binary. This has the nice side effect of fixing everything else in memory, making it easier for our instrumentation scripts to work without having to calculate new addresses for each execution. Patching the executable to disable relocation is straightforward. In the PE header for the executable, there is a flag in `IMAGE_NT_HEADERS` > `IMAGE_OPTIONAL_HEADER` called `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE`. Viewed in PE View, we can see that in this sample, that bit is set:



```
00000156    8140    DLL Characteristics
                    0040    IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
                    0100    IMAGE_DLLCHARACTERISTICS_NX_COMPAT
                    8000    IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE
```

This bit can be un-set in a hex editor at the offset 0x156 into the executable file.

Installation and Persistence

WinMain in this sample begins at 0x00408F90, and branches early, at 0x00408FEA at a test following a function (0x00408AB0, IDB: `check_installed`) that checks to see if the malware is currently installed. Prior to this, a function (0x00401AF0, IDB: `setup_strings`) is called to set up string objects for the rest of execution. That these objects are all created at runtime with the actual C-style strings from the data segment is part of what complicates static analysis.

The strings form three groups forming what passes for the configuration of this malware:

- A command and control server string ("priv8darkshop.com" in this sample)
- A list of executable file names that will be used as the installed sample's name (Starting with "jusched.exe")

    - It has been observed at runtime that if a file of its chosen name already exists at the install location, the next filename on the list is chosen
- A list of process names that will not be searched for track data later on in the main loop.

If the install check fails, the main loop does not execute. Instead, the code responsible for installation and persistence begins at the basic block located at 0x00408FF0. The logged-in user's `AppData\Roaming` folder is located, a directory named "Java SE Platform Updater" is created, and the malware is copied to that location. Registry persistence is established in:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\
```

Once installed, the new copy is run with the original as a command-line argument. When the install check passes on the new run, the original is deleted (after a brief `Sleep()` call in order to give it time to exit) at 0x0040921B.

For our purposes, it would be much more convenient to simply run the malware from our local working directory and have it bypass most of the installation, persistence, and checking. For that reason, some of the earliest actions taken in our `jackpos_harness.py` are:

- patch_cnc at 0x00401B38 - changing the command and control server to one under our control
- patch_install_check at 0x00408FE8 - forces execution of the main loop, rather than the installation code, even if the installation check fails
- shell_execute_blocker at 0x0040B796 - prevents call to ShellExecute that would run the watchdog process. Also serves to prevent the command and control server from executing anything on the infected client later on.

Instrumenting Command and Control Communication

A function at 0x00403710 (IDB: cnc_checkin) executes once before entering the main loop, and once on each iteration of the main loop. The purpose of this function is to:

- (in 0x00403F10, IDB: post_echo) - Test for the presence of the command and control server by retrieving the path /post/echo from its web server. It can be seen in the PHP source of the command and control (in protected/controllers/PostController.php) that it responds “up” to this request. This function tests for that response, and only if it passes this (very simple) check, does it proceed to communicate further with the command and control. Presumably this is to keep the sample from “giving away” too much in a sandbox environment.
- (beginning at the basic block at 0x00403859) - Crafts POST data for checking into the command and control is crafted from:
  - Base64 Encoded (0x00401000, IDB: base_64_encode) track data retrieved from processes, if any
  - MAC address information (0x0040CD10, IDB: mac_addr) used to uniquely identify an individual bot within the command and control.
- (in 0x0040CD10, IDB: cnc_post_data) POST data is sent to the command and control server, and if there is a command in the command and control queue for that particular bot, the malware receives the command as a response.

We can observe this in action with the following jackpos_harness.py functions:

- cnc_online at 0x0040320A - Observes the /post/echo check
  - cnc_online_end at 0x00403321 - Displays the results of the check
- cnc_send at 0x004035BC - Displays the data (MAC, and track data, if any) being sent to the command and control

- cnc_recv at 0x00403627 - Displays data being sent back to the malware from the command and control server (If anything, presumably a command to be parsed and followed).

Identifying Commands

The focus of this analysis is on the theft of credit card track information from infected hosts, but in our jackpos_harness.py we at least point out where the three commands that the command and control server can issue begin to be parsed. Much of this processing in the malware takes place in the WinMain function.

- kill (harness: process_kill) - removes the malware's persistence and kills the process.
- update (harness: process_update) - causes the malware replaces the current installation of JackPOS with the latest version (as returned from the path /post/download). Updated malware is kept on the command and control server in /clients/.
- exec <url> (harness: process_exec) - causes the malware to download and run from a url (rather than replacing the current malware as in the "update" command). URLs provided by the "official" command and control server involve files in the /exec/ directory.

Reading Memory from Processes

In order to effectively steal credit cards from many different point of sale systems, without being familiar with the internals of any specific one, and supporting a wide variety of different software, the technique JackPOS adopts is one of scanning memory for credit card track data. In the function at 0x0040A9D0 (IDB: get_processes), the list of available processes is walked, and a list of processes to be scanned is built, avoiding 64-bit processes, and those matching names from the list built in the configuration string setup (mostly system processes).

Once the list of processes is built, for each process the function at 0x00407CC0 (IDB: search_process_memory) is called, which iterates through each region of memory for a process that the process is allowed to read. Each memory region is then passed through two functions that search for credit card data.

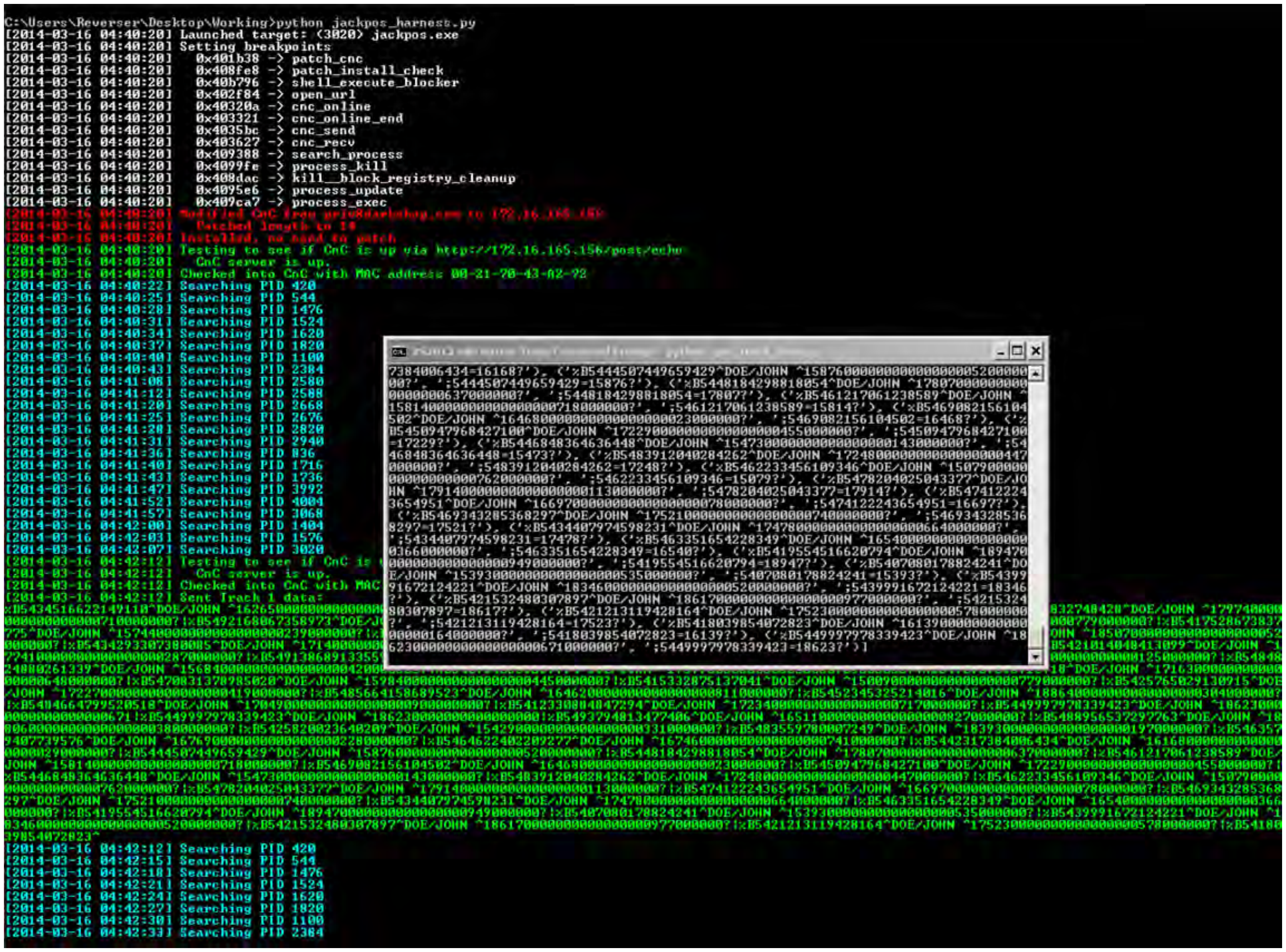
jackpos_harness.py identifies each call to search_process_memory with the search_process callback. A separate harness, search_proc_harness.py, can be used to scan arbitrary process ID's from the command line without invoking the rest of the malware. This allows for easily testing the process reading and track data matching functionality against our data generation scripts, without having to wait for the malware to scan back around to the desired PID.

Matching Credit Card Track Data

Functions at 0x00407E30 (IDB: look_for_track1_data) and 004081F0 (IDB: look_for_track2_data) seek out credit card magnetic stripe data in regions of memory, given starting and ending addresses. The magnetic stripe data on the back of physical credit cards is encoded in a standard known as ISO/IEC 7813[12,13]. We can use this information to inform our analysis of these two regular-expression-like functions, which would otherwise be tedious without prior knowledge of the format. Code is provided in gen_track_data.py to generate "fake" track data and hold it in memory for JackPOS to find and submit back to the command and control server. With this code we can demonstrate how to successfully navigate these two functions.

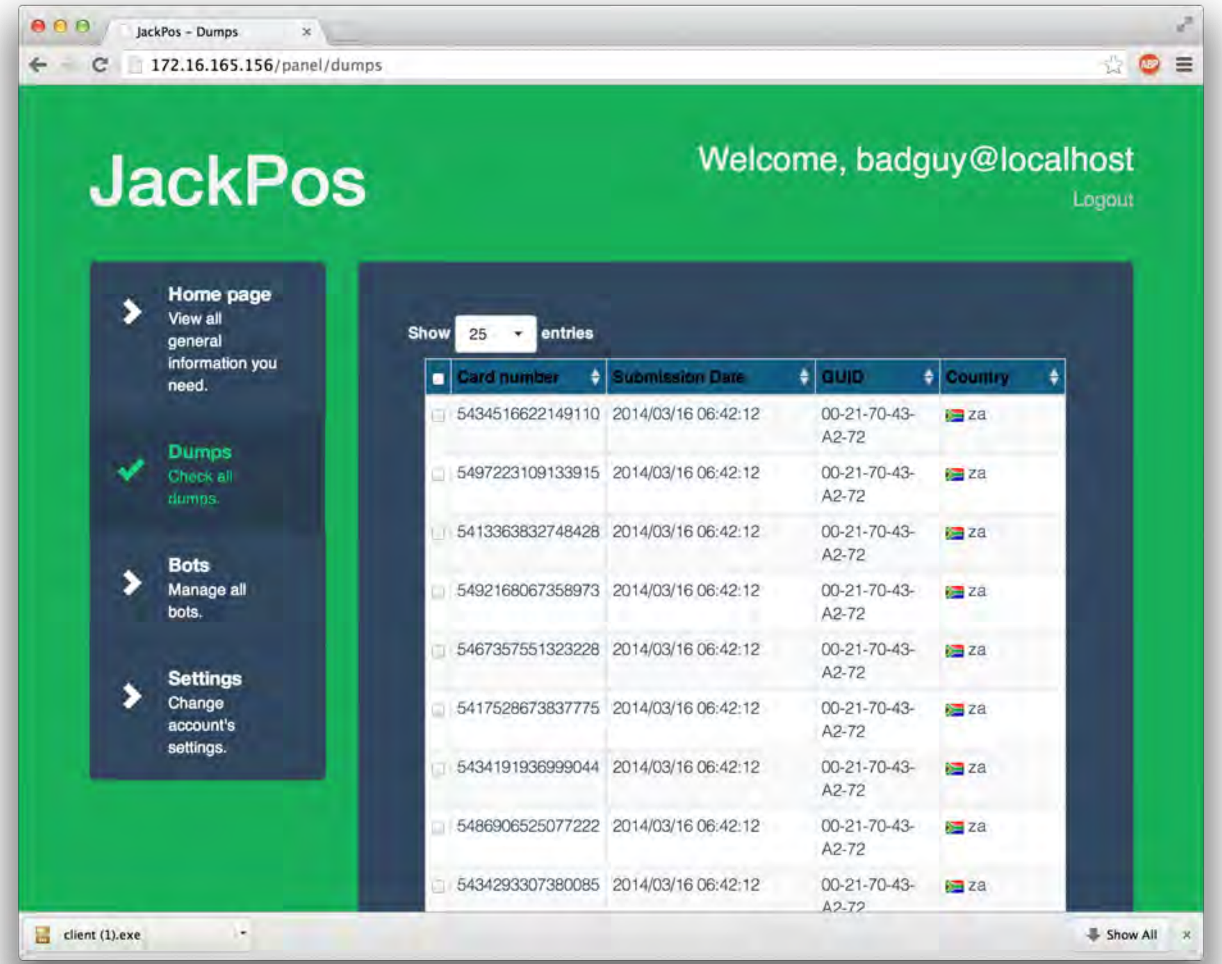
Analysis Wrap-Up

Combined, track data generation and instrumentation of JackPOS allows us to watch the execution of this malware in a visual way, which will be demonstrated during the talk. The following screenshots illustrates track data being found and submitted successfully to the command and control server:



```
C:\Users\Reverser\Desktop\Working>python jackpos_harness.py
[2014-03-16 04:40:20] Launched target: (3020) jackpos.exe
[2014-03-16 04:40:20] Setting breakpoints
[2014-03-16 04:40:20]   0x401b38 -> patch_cnc
[2014-03-16 04:40:20]   0x408fe8 -> patch_install_check
[2014-03-16 04:40:20]   0x40b796 -> shell_execute_blocker
[2014-03-16 04:40:20]   0x402f84 -> open_url
[2014-03-16 04:40:20]   0x40320a -> cnc_online
[2014-03-16 04:40:20]   0x403321 -> cnc_online_end
[2014-03-16 04:40:20]   0x4035bc -> cnc_send
[2014-03-16 04:40:20]   0x403627 -> cnc_recv
[2014-03-16 04:40:20]   0x409388 -> search_process
[2014-03-16 04:40:20]   0x4099fe -> process_kill
[2014-03-16 04:40:20]   0x408dac -> kill__block_registry_cleanup
[2014-03-16 04:40:20]   0x4095e6 -> process_update
[2014-03-16 04:40:20]   0x409ca7 -> process_exec
[2014-03-16 04:40:20] Modified CnC from priv8darkshop.com to 172.16.165.156
[2014-03-16 04:40:20]   Patched length to 14
[2014-03-16 04:40:20] Installed, no need to patch
[2014-03-16 04:40:20] Testing to see if CnC is up via http://172.16.165.156/post/echo
[2014-03-16 04:40:20]   CnC server is up.
[2014-03-16 04:40:20] Checked into CnC with MAC address 00-21-70-43-A2-72
[2014-03-16 04:40:22] Searching PID 420
[2014-03-16 04:40:25] Searching PID 544
[2014-03-16 04:40:28] Searching PID 1476
[2014-03-16 04:40:31] Searching PID 1524
[2014-03-16 04:40:34] Searching PID 1620
[2014-03-16 04:40:37] Searching PID 1820
[2014-03-16 04:40:40] Searching PID 1100
[2014-03-16 04:40:43] Searching PID 2384
[2014-03-16 04:41:08] Searching PID 2580
[2014-03-16 04:41:12] Searching PID 2588
[2014-03-16 04:41:20] Searching PID 2668
[2014-03-16 04:41:25] Searching PID 2676
[2014-03-16 04:41:28] Searching PID 2820
[2014-03-16 04:41:31] Searching PID 2940
[2014-03-16 04:41:36] Searching PID 836
[2014-03-16 04:41:40] Searching PID 1716
[2014-03-16 04:41:43] Searching PID 1736
[2014-03-16 04:41:47] Searching PID 3992
[2014-03-16 04:41:52] Searching PID 4004
[2014-03-16 04:41:57] Searching PID 3068
[2014-03-16 04:42:00] Searching PID 1404
[2014-03-16 04:42:03] Searching PID 1576
[2014-03-16 04:42:07] Searching PID 3020
[2014-03-16 04:42:12] Testing to see if CnC is
[2014-03-16 04:42:12]   CnC server is up.
[2014-03-16 04:42:12] Checked into CnC with MAC
[2014-03-16 04:42:12] Sent Track 1 data:
%B5434516622149110^DOE/JOHN ^16265000000000000                                      832748420^DOE/JOHN ^1797400000
00000000000000710000000?!%B5492168067358973^DOE/J                                   000779000000?!%B5417528673837
775^DOE/JOHN ^15744000000000000000239000000?!%                                      OHN ^185070000000000000000052
000000?!%B5434293307380085^DOE/JOHN ^1714000000                                     B5421014048413099^DOE/JOHN ^1
7241000000000000000000287000000?!%B5491386891335                                    0000000000125000000?!%B54848
2480261339^DOE/JOHN ^156840000000000000000042000                                    18^DOE/JOHN ^1716300000000000
000006480000000?!%B5470831370785020^DOE/JOHN ^159840000000000000000445000000?!%B5415332875137041^DOE/JOHN ^15009000000000000000000779000000?!%B5425765029130915^DOE
/JOHN ^17227000000000000000000419000000?!%B5485664158689523^DOE/JOHN ^16462000000000000000000811000000?!%B5452345325214016^DOE/JOHN ^18864000000000000000000304000000?
!%B5404664799520518^DOE/JOHN ^17049000000000000000000900000000?!%B5412330804847294^DOE/JOHN ^17234000000000000000000717000000?!%B5449997978339423^DOE/JOHN ^18623000
000000000000000671!%B5449997978339423^DOE/JOHN ^186230000000000000000000!%B5493794813477406^DOE/JOHN ^16511000000000000000000827000000?!%B5488956537297763^DOE/JOHN ^18
8060000000000000000000380000000?!%B5425820023640209^DOE/JOHN ^15429000000000000000000331000000?!%B5483559780007249^DOE/JOHN ^18393000000000000000000197000000?!%B546357
9407739576^DOE/JOHN ^16769000000000000000000228000000?!%B5464622402209277^DOE/JOHN ^16746000000000000000000741000000?!%B5442317384006434^DOE/JOHN ^161680000000000000
0000029000000?!%B5444507449659429^DOE/JOHN ^15876000000000000000000520000000?!%B5448184298818054^DOE/JOHN ^17807000000000000000000637000000?!%B5461217061238589^DOE/
JOHN ^15814000000000000000000718000000?!%B5469082156104502^DOE/JOHN ^16468000000000000000000230000000?!%B5450947968427100^DOE/JOHN ^17229000000000000000000455000000?!
%B5446848364636448^DOE/JOHN ^15473000000000000000000143000000?!%B5483912040284262^DOE/JOHN ^17248000000000000000000447000000?!%B5462233456109346^DOE/JOHN ^1507900000
000000000000000762000000?!%B5478204025043377^DOE/JOHN ^17914000000000000000000113000000?!%B5474122243654951^DOE/JOHN ^16697000000000000000000780000000?!%B5469343285368
297^DOE/JOHN ^17521000000000000000000740000000?!%B5434407974598231^DOE/JOHN ^17478000000000000000000664000000?!%B5463351654228349^DOE/JOHN ^16540000000000000000000366
000000?!%B5419554516620794^DOE/JOHN ^18947000000000000000000949000000?!%B5407080178824241^DOE/JOHN ^15393000000000000000000535000000?!%B5439991672124221^DOE/JOHN ^1
8346000000000000000000520000000?!%B5421532480307897^DOE/JOHN ^18617000000000000000000977000000?!%B5421213119428164^DOE/JOHN ^17523000000000000000000578000000?!%B54180
39854072823^
[2014-03-16 04:42:12] Searching PID 420
[2014-03-16 04:42:15] Searching PID 544
[2014-03-16 04:42:18] Searching PID 1476
[2014-03-16 04:42:21] Searching PID 1524
[2014-03-16 04:42:24] Searching PID 1620
[2014-03-16 04:42:27] Searching PID 1820
[2014-03-16 04:42:30] Searching PID 1100
[2014-03-16 04:42:33] Searching PID 2384
```

```
7384006434=16168?'), ('%B5444507449659429^DOE/JOHN ^158760000000000000000000520000000
00?', ';5444507449659429=15876?'), ('%B5448184298818054^DOE/JOHN ^1780700000000000
00000000637000000?', ';5448184298818054=17807?'), ('%B5461217061238589^DOE/JOHN ^
15814000000000000000000718000000?', ';5461217061238589=15814?'), ('%B5469082156104
502^DOE/JOHN ^16468000000000000000000230000000?', ';5469082156104502=16468?'), ('%
B5450947968427100^DOE/JOHN ^17229000000000000000000455000000?', ';5450947968427100
=17229?'), ('%B5446848364636448^DOE/JOHN ^15473000000000000000000143000000?', ';54
46848364636448=15473?'), ('%B5483912040284262^DOE/JOHN ^17248000000000000000000447
000000?', ';5483912040284262=17248?'), ('%B5462233456109346^DOE/JOHN ^1507900000
000000000000762000000?', ';5462233456109346=15079?'), ('%B5478204025043377^DOE/JO
HN ^17914000000000000000000113000000?', ';5478204025043377=17914?'), ('%B547412224
3654951^DOE/JOHN ^16697000000000000000000780000000?', ';5474122243654951=16697?'),
 ('%B5469343285368297^DOE/JOHN ^17521000000000000000000740000000?', ';546934328536
8297=17521?'), ('%B5434407974598231^DOE/JOHN ^17478000000000000000000664000000?',
';5434407974598231=17478?'), ('%B5463351654228349^DOE/JOHN ^1654000000000000000000
0366000000?', ';5463351654228349=16540?'), ('%B5419554516620794^DOE/JOHN ^189470
00000000000000000949000000?', ';5419554516620794=18947?'), ('%B5407080178824241^DO
E/JOHN ^15393000000000000000000535000000?', ';5407080178824241=15393?'), ('%B54399
91672124221^DOE/JOHN ^18346000000000000000000520000000?', ';5439991672124221=18346
?'), ('%B5421532480307897^DOE/JOHN ^18617000000000000000000977000000?', ';54215324
80307897=18617?'), ('%B5421213119428164^DOE/JOHN ^17523000000000000000000578000000
?', ';5421213119428164=17523?'), ('%B5418039854072823^DOE/JOHN ^1613900000000000
00000164000000?', ';5418039854072823=16139?'), ('%B5449997978339423^DOE/JOHN ^18
623000000000000000000671000000?', ';5449997978339423=18623?')]
```


JackPos - Dumps
172.16.165.156/panel/dumps
JackPos
Welcome, badguy@localhost
Logout
Home page
View all general information you need.
Dumps
Check all dumps.
Bots
Manage all bots.
Settings
Change account's settings.
client (1).exe
Show All

Show 25 entries

Card number	Submission Date	GUID	Country
5434516622149110	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5497223109133915	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5413363832748428	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5492168067358973	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5467357551323228	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5417528673837775	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5434191936999044	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5486906525077222	2014/03/16 06:42:12	00-21-70-43-A2-72	za
5434293307380085	2014/03/16 06:42:12	00-21-70-43-A2-72	za

CONCLUSIONS

By addressing the lack of reproducibility and verifiability in published malware analyses, we have the potential to more effectively illustrate the inner workings of malware to lay audiences, fellow malware researchers, and students of the field alike. While factors exist that might dissuade an organization or individual researcher from publishing a reproducible analysis, the population of malware analysts and other information security professionals that consume analyses should make it known that they expect and appreciate more “complete” analyses. Those that publish analyses should be encouraged to become more accountable to peer review, and analyses that meet expectations of reproducibility and verifiability should be recommended by industry professionals over those that do not.

Virtualization of resources needed to give malware the environment and data that it "wants" in order to carry out its most interesting code path, implementation of network services needed (such as command-and-control services), and programmatic instrumentation of the malicious software itself, all combine to illustrate the operation of malware samples in a way that is more instructive and interactive. Analysts that wish to take published analyses further can use reproducible analyses as a basis to work from, rather than starting "from scratch". Future work in this area should explore the potential for implementing cloud-based services for publishing instrumented analyses in a way that allows an another analyst to observer and work with malware samples in a more immediate way.

BIBLIOGRAPHY

1. Stodden, Victoria C., "The Scientific Method in Practice: Reproducibility in the Computational Sciences", MIT Sloan School of Management Working Papers, 2010, http://academiccommons.columbia.edu/item/ac:140117
2. Sikorsky, Michael and Honig, Andrew, *Practical Malware Analysis*, No Starch Press, 2013
3. Journal of Computational Science, http://www.journals.elsevier.com/journal-of-computational-science/
4. Executable Paper Grand Challenge, http://www.executablepapers.com/about-challenge.html
5. Pieter Van Gorp, Steffen Mazanek, "SHARE: a web portal for creating and sharing executable research papers", Procedia Computer Science, Volume 4, 2011, Pages 589-597, ISSN 1877-0509, http://dx.doi.org/10.1016/j.procs.2011.04.062.

6. Munroe, Randall, “Network”, XKCD 350, http://xkcd.com/350/.
7. Grunzweig, Josh, Trustwave SpiderLabs, “JackPOS - The House Always Wins”, February 11, 2014, http://blog.spiderlabs.com/2014/02/jackpos-the-house-always-wins.html
8. unixfreakjp, Malware Must Die!, “Cyber Intelligence: The JackPOS Behind the Screen”, February 13, 2014, http://blog.malwaremustdie.org/2014/02/cyber-intelligence-jackpos-behind-screen.html
9. @xylit0l, xylitol@malwareint.com, “CyberCrime Tracker”, http://cybercrime-tracker.net
10. Vilas, Mario, WinAppDbg, http://winappdbg.sourceforge.net/
11. Yii Framework, http://www.yiiframework.com/
12. Wikipedia, “ISO/IEC 7813”, http://en.wikipedia.org/wiki/ISO/IEC_7813
13. Padilla, L., “Magnetic Stripe Examples: Standard Cards”, http://www.gae.ucm.es/~padilla/extrawork/magexam1.html