Toward Automated Dynamic Malware Analysis Using CWSandbox

The authors present CWSandbox, which executes malware samples in a simulated environment, monitors all system calls, and automatically generates a detailed report to simplify and automate the malware analyst's task.

Carsten Willems, Thorsten Holz, and Felix Freiling
University of Mannheim, Germany

Malware is notoriously difficult to combat because it appears and spreads so quickly. Most security products such as virus scanners look for signatures—characteristic byte sequences—to identify malicious code. Malware, however, has adapted to that approach. Poly- or metamorphic worms avoid detection by changing their appearance, for example, whereas flash worms stealthily perform reconnaissance without infecting vulnerable machines, waiting to pursue strategic spreading plans that can infect thousands of machines within seconds.

In the face of such automated threats, security researchers can't combat malicious software using manual methods of disassembly or reverse engineering. Therefore, analysis tools must analyze malware automatically, effectively, and correctly. Automating this process means that the analysis tool should create detailed reports of malware samples quickly and without user intervention. Analysts could then use the machine-readable reports to initiate automated responses—automatically updating an intrusion detection system's signatures, for example, and protecting networks from new malware on the fly. An effective analysis tool must log the malware's relevant behaviors—the tool shouldn't overlook any of the executed functionality because analysts will use the information to realistically assess the threat. Finally, the tool should correctly analyze the malware—the sample should initiate every logged action to avoid false positives.

In this article, we describe the design and implementation of CWSandbox, a malware analysis tool that fulfills our three design criteria of automation, effectiveness, and correctness for the Win32 family of operating systems. We show how to use API hooking and dynamic linked library (DLL) injection techniques to implement the necessary rootkit functionality to avoid detection by the malware. We acknowledge that these techniques aren't new; however, we've assembled the techniques in a unique combination that provides a fully functional, elegantly simple, and arguably powerful automated malware analysis tool.

Behavior-based malware analysis

Combining dynamic malware analysis, API hooking, and DLL injection within the CWSandbox lets analysts trace and monitor all relevant system calls and generates an automated, machine-readable report that describes

- the files the malware sample created or modified;
- the changes the malware sample performed on the Windows registry;
- which DLLs the malware loaded before execution;
- which virtual memory areas it accessed;
- the processes that it created;
- the network connections it opened and the information it sent; and
- other information, such as the malware's access to protected storage areas, installed services, or kernel drivers.

CWSandbox's reporting features aren't perfect—that is, it reports only the malware's visible behavior and not how it's programmed, and using the CWSandbox might cause some harm to other machines connected to the network. Yet, the information derived from the CWSandbox for even the shortest of time periods is sur-

prisingly rich; in most cases, it's more than sufficient to assess the danger originating from malware.

In the following paragraphs, we introduce the individual building blocks and techniques behind CWSandbox.

Dynamic malware analysis

Dynamic analysis observes malware behavior and analyzes its properties by executing the malware in a simulated environment—in our case, the sandbox. Two different approaches to dynamic malware analysis exist, each resulting in different granularity and quality:

- taking an image of the complete system state before malware execution and comparing it to the complete system state after execution; and
- monitoring the malware's actions during execution with the help of a specialized tool, such as a debugger.

The first option is easier to implement but delivers more coarse-grained results, which sometimes are sufficient to gain an overview of what a given binary does. This approach analyzes only the malware's cumulative effects without taking into account dynamic changes—such as the malware generating a file during execution and deleting it before termination, for example. The second approach is harder to implement, but we chose to use it in the CWSandbox because it delivers much more detailed results.

Dynamic analysis has a drawback: it analyzes only a single malware execution at a time. In contrast, static malware analysis analyzes the source code, letting analysts observe all possible malware executions at once. Static analysis, however, is rather difficult to perform because the malware's source code isn't usually available. Even if it is, you can never be sure that undocumented modifications of the binary executable didn't occur. Additionally, static analysis at the machine code level can be extremely cumbersome because malware often uses code-obfuscation techniques such as compression, encryption, or self-modification to evade decompilation and analysis.

API hooking

Programmers use the Windows API to access system resources such as files, processes, network information, the registry, and other Windows areas. Applications use the API rather than making direct system calls, offering the possibility for dynamic analysis if we can monitor the relevant API calls and their parameters. The Windows system directory contains the API, which consists of several important DLLs, including `kernel32.dll`, `ntdll.dll`, `ws2 32.dll`, and `user32.dll`.

To observe a given malware sample's control flow, we need to access the API functions. One possible way to achieve this is by *hooking*—intercepting a call to a function. When an application calls a function, it's rerouted to a different location where customized code—the hook or hook function—resides. The hook then performs its own operations and transfers control back to the original API function or prevents its execution completely. If hooking is done properly, it's hard for the calling application to detect the hooked API function or that it's called instead of the original function. In our case, the malware could try to detect the hooking function, so we must carefully implement it and try to hide the analysis environment from the malware process as much as possible.

Several different methods let hook functions intercept system calls from potentially malicious user applications on their way to the kernel.[1] For example, you can intercept the execution chain either inside the user process in one or multiple parts of the Windows API or inside the Windows kernel by modifying the interrupt descriptor table (IDT) or the system service dispatch table (SSDT). Other methods have different advantages, disadvantages, and complexity. We use in-line code overwriting because it's one of the more effective and efficient methods.

In-line code overwriting directly overwrites the DLL's API function code that's loaded into the process memory. Therefore, calls to the API functions are rerouted to the hook function, regardless of when they occur or whether they're linked implicitly or explicitly. Implicit linking occurs when an application's code calls an exported DLL function, whereas applications must make a function call to explicitly load the DLL at runtime with explicit linking. We can overwrite the function code using the following steps:

1. Create a target application in suspended mode. Windows loads and initializes the application and all implicitly linked DLLs, but it doesn't start the main thread so the target application doesn't perform any operations.
2. When the initialization work is done, CWSandbox looks up every to-be-hooked function in the DLL's export address table (EAT) and retrieves their code entry points.

In the face of such threats, security researchers can't combat malicious software using manual methods of disassembly or reverse engineering.

3. Save the original code in advance so you can later reconstruct the original API function.
4. Overwrite the first few instructions of each API function with a `JMP` (or a `call`) instruction that

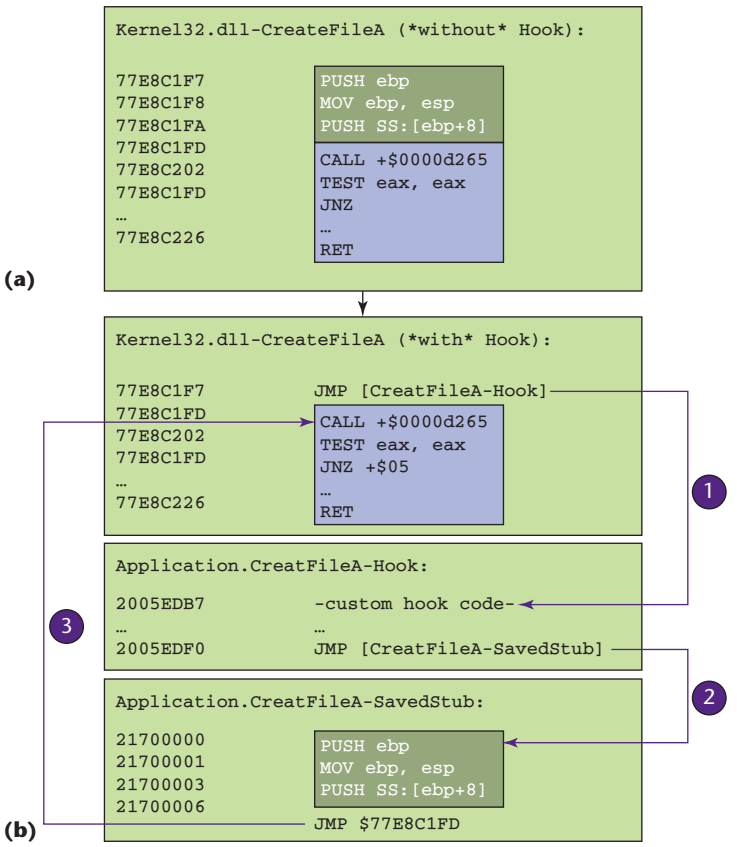

Figure 1. In-line code overwriting. (a) shows the original function code. In (b), the `JMP` instruction overwrites the API function's first block (1) and transfers control to our hook function whenever the to-be-analyzed application calls the API function. (2) The hook function performs the desired operations and then calls the original API function's saved stub. (3) The saved stub performs the overwritten instructions and branches to the original API function's unmodified part.

leads to the hook function.

5. To complete the process, hook the `LoadLibrary` and `LoadLibraryEx` API functions, which allow the explicit binding of DLLs.

If an application loads the DLL dynamically at runtime, you can use this same procedure to overwrite the function entry points. The CWSandbox carries out these steps in the initialization phase to set up the hooking functions.

Figure 1a shows the original function code for `CreateFileA`, which is located in `kernel32.dll`. The instructions are split into two blocks: the first marks the block that we'll overwrite to delegate control to our hook function; the second block includes the instructions that the API hook won't touch. Figure 1b shows the situation after we installed the hook. We overwrite the first six bytes of each to-be-analyzed function with a `JMP` instruction to our hook code. In the hook function, we can save the called API function's parameters or modify them if necessary. Then, we execute the bytes that we overwrote in the first phase and then `JMP` back to execute the rest of the API function. There's no need to call it with a `JMP` instruction: the hook function can call the original API with a `CALL` operation and regain control when the called API function performs the `RET`. The hook function then analyzes the result and modifies it if necessary. Holy Father offers one of the most popular and detailed descriptions of this approach,[2] and Microsoft also offers a library called Detours for this purpose (www.research.microsoft.com/sn/detours).

To offer a complete hooking overview, we must mention system service hooking, which occurs at a lower level within the Windows operating system and isn't considered to be API hooking. Two additional possibilities exist for rerouting API calls: we can modify an entry in the IDT such that `Int 0x2e`, which is used for invoking system calls, points to the hooking routine, or we can manipulate the entries in the SSDT so that the system calls can be intercepted depending on the service IDs. We don't use these techniques because API hooking is much easier to implement and delivers more accurate results. In the future, we might extend CWSandbox to use kernel hooks because they're more complicated to detect.

On a side note, programs that directly call the kernel to avoid using the Windows API can bypass API hooking techniques. However, this is rather uncommon because the malware author must know the target operating system, its service pack level, and other information in advance. Our results show that most malware authors design autonomous-spreading malware to attack large user bases, so they commonly use the Windows API.

DLL code injection

DLL code injection lets us implement API hooking in a modular and reusable way. However, API hooking with inline code overwriting makes it necessary to patch the application after it has been loaded into memory. To be successful, we must copy the hook functions into the target application's address space so they can be called from within the target—this is the actual code injection—and bootstrap the API hooks in the target application's address space using a specialized thread in the malware's memory.

How can we insert the hook functions into the process running the malware sample? It depends on the hooking method we use. In any case, we have to manipulate the target process's memory—changing the application's import address table (IAT), changing the loaded DLLs' export address table (EAT), or directly overwriting the API function code. In Windows, we can implant and install API hook functions by accessing another process's

virtual memory and executing code in a different process's context.

Windows `kernel32.dll` offers the API functions `ReadProcessMemory` and `WriteProcessMemory`, which lets the CWSandbox read and write to an arbitrary process's virtual memory, allocating new memory regions or changing an already allocated memory region's using the `VirtualAllocEx` and `VirtualProtectEx` functions.

It's possible to execute code in another process's context in at least two ways:

- suspend one of the target application's running threads, copy the to-be-executed code into the target's address space, set the resumed thread's instruction pointer to the copied code's location, and then resume the thread; or
- copy the to-be-executed code into the target's address space and create a new thread in the target process with the code location as the start address.

With these building blocks in place, it's now possible to inject and execute code into another process.

The most popular technique is DLL injection, in which the CWSandbox puts all custom code into a DLL and the hook function directs the target process to load this DLL into its memory. Thus, DLL injection fulfills both requirements for API hooking: the custom hook functions are loaded into the target's address space, and the API hooks are installed in the DLL's initialization routine, which the Windows loader calls automatically.

The API functions `LoadLibrary` or `LoadLibraryEx` perform the explicit DLL linking; the latter allows more options, whereas the first function's signature is very simple—the only parameter it needs is a pointer to the DLL name.

The trick is to create a new thread in the target process's context using the `CreateRemoteThread` function and then setting the code address of the API function `LoadLibrary` as the newly created thread's starting address. When the to-be-analyzed application executes the new thread, the `LoadLibrary` function is called automatically inside the target's context. Because we know `kernel32.dll`'s location (always loaded at the same memory address) from our starter application, and know the `LoadLibrary` function's code location, we can also use these values for the target application.

CWSandbox architecture

With the three techniques we described earlier set up, we can now build the CWSandbox system that's capable of automatically analyzing a malware sample. This system outputs a behavior-based analysis; that is, it executes the malware binary in a controlled environment so that we can observe all relevant function calls to the Windows API, and generates a high-level summarized report from the monitored API calls. The report provides data for each process and its associated actions—one subsection for all accesses to the file system and another for all network operations, for example. One of our focuses is on bot analysis, so we spent considerable effort on extracting and evaluating the network connection data.

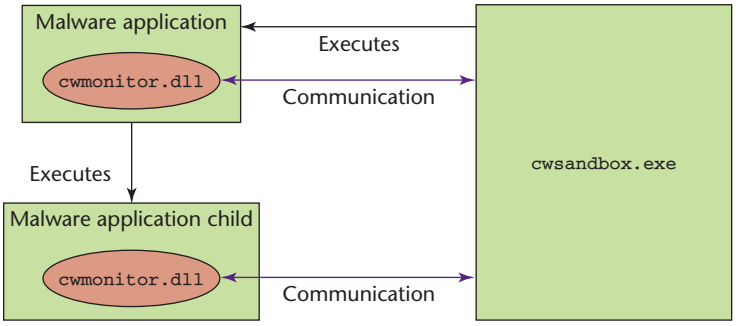

Figure 2. CWSandbox overview. `CWSandbox.exe` creates a new process image for the to-be-analyzed malware binary and then injects the `cwmonitor.dll` into the target application's address space. With the help of the DLL, we perform API hooking and send all observed behavior via the communication channel back to `cwsandbox.exe`. We use the same procedure for child or infected processes.

After it analyzes the API calls' parameters, the sandbox routes them back to their original API functions. Therefore, it doesn't block the malware from integrating itself into the target operating system—copying itself to the Windows system directory, for example, or adding new registry keys. To enable fast automated analysis, we execute the CWSandbox in a virtual environment so that the system can easily return to a clean state after completing the analysis process. This approach has some drawbacks—namely, detectability issues and slower execution—but using CWSandbox in a native environment such as a normal commercial off-the-shelf system with an automated procedure that restores the system to a clean state can help circumvent these drawbacks.

The CWSandbox has three phases: initialization, execution, and analysis. We discuss each phase in more detail in the following sections.

Initialization phase

In the initialization phase, the sandbox, which consists of the `cwsandbox.exe` application and the `cwmonitor.dll` DLL, sets up the malware process. This DLL installs the API hooks, realizes the hook functions, and exchanges runtime information with the sandbox.

The DLL's life cycle is also divided into three phases: initialization, execution, and finishing. The DLL's main function is to handle the first and last phases; the hook functions handle the execution phase. DLL operations are executed during the initialization and finishing

Related work in malware behavior analysis

Several tools exist for automatically analyzing malicious software behaviors. Despite some similarities, our CWSandbox has the advantage of generating a detailed, behavior-based analysis report and automating the whole process to a high degree.

The Norman SandBox (http://sandbox.norman.no) simulates an entire computer and a connected network by reimplementing the core Windows system and executing the malware binary within the simulated environment. It's also possible to execute the malware binary with a live Internet connection. The company's Web site features implementation details, a description of the underlying technology, and a live demo. Such environments are mostly transparent to the malware, which can't detect that they're being executed within a simulated environment. Yet, simulations don't let the malware processes interfere with, infect, or modify other running processes because no other processes run within the simulation. By not monitoring this interference, valuable information might be missed. By using a real operating system as CWSandbox's base, we allow the malware samples to interfere with the system with only the limited disturbance created by API hooking.

Another comparable approach is TTAnalyze.[1] Like our sandbox, TTAnalyze uses API hooking, but it differs from our solution in basically one area: it uses the PC emulator QEMU[2] rather than virtual machines, which makes it harder for the malware to detect that it's running in a controlled environment (although it means no significant difference for the analysis).

A different approach is Chas Tomlin's Litterbox (www.wiul.org), in which malware is executed on a real Windows system, rather than a simulated or emulated one. After 60 seconds of execution, the host machine is rebooted and forced to boot from a Linux image. After booting Linux, Litterbox mounts the Windows partition and extracts the Windows registry and complete file list; the Windows partition reverts back to its initial clean state. Litterbox focuses on network activity, so it makes several dispositions of the simulated network. During malware execution, the Windows host connects to a virtual Internet with an IRC server running, which answers positively to all incoming IRC connection requests. The tool captures all packets to examine all other network traffic afterwards. This approach is advantageous to CWSandbox because IRC connections are always successful, whereas CWSandbox encounters malware binaries whose associated C&C server is already mitigated. However, because Litterbox takes only a snapshot of the infected system, it can't monitor dynamic actions such as the creation of new processes. The Reusable Unknown Malware Analysis Net (Truman; www.lurhq.com) takes a similar approach.

Galen Hunt and Doug Brubacher introduced Detours, a library for instrumenting arbitrary Windows functions.[3] We opted to implement our own API hooking mechanism to have greater flexibility and more control over the instrumented functions, but this library makes it possible to implement an automated approach to malware analysis that is similar to CWSandbox.

The concepts of system call sequence analysis and API hooking are well-known in the area of intrusion detection. A typical approach includes a training phase in which the IDS system observes system calls of the complete system or specific processes and creates a profile of "normal" behavior. During operation, the system call sequences are compared against this profile; upon detecting a deviation, the system sounds an alarm that indicates an anomaly. Stephanie Forrest and her coauthors give one of the earliest descriptions of this approach,[4] and Steven Hofmeyr and colleagues introduced a method for detecting intrusions at the privileged processes level.[5] System call sequence monitoring can also facilitate process confinement as introduced with Systracer by Provos.[6] Within CWSandbox, we use system call sequence analysis to observe the behavior of malware processes and construct detailed reports by correlating the collected data.

References

1. U. Bayer, C. Kruegel, and E. Kirda, "TTanalyze: A Tool for Analyzing Malware," *Proc. 15th Ann. Conf. European Inst. for Computer Antivirus Research* (EICAR), *EICAR Conf. Proceedings*, 2006, pp. 180–192.
2. F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," *Proc. Usenix 2005 Ann. Technical Conf.* (Usenix '05), Usenix Assoc., 2005, pp. 41–46.
3. G.C. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," *Proc. 3rd Usenix Windows NT Symp.*, Usenix Assoc., 1999, pp. 135–143.
4. S. Forrest et al., "A Sense of Self for Unix Processes," *Proc. 1996 IEEE Symp. Security and Privacy* (S&P 1996), IEEE CS Press, 1996, pp. 120–128.
5. S.A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *J. Computer Security*, vol. 6, no. 3, 1998, pp. 151–180.
6. N. Provos, "Improving Host Security with System Call Policies," *Proc. 12th Usenix Security Symp.* (Security '03), Usenix Assoc., 2003, pp. 257–271.

phases and every time one of the hooked API functions is called. Additionally, the DLL informs the sandbox when the malware starts a new process or injects code into a running process. As Figure 2 shows, the sandbox then injects a new instance of the DLL into the newly created or existing process so that it captures all API calls from this process.

Execution phase

If everything initializes correctly, malware processing resumes and the execution phase starts. Otherwise, the sandbox kills the newly created malware process and terminates. During the malware's execution, the sandbox reroutes the hooked API calls to the referring hook functions in the DLL, which inspects the call parameters, informs the sandbox about the API calls in the form of notification objects, and then delegates control to the

original function or returns directly to the application performing the API call, depending on the type of API function called. After the original API call returns, the hook function inspects the result and might modify it before returning to the calling malware application.

Because the malware sample shouldn't be aware that it's being executed inside a controlled environment, the `cwmonitor.dll` implements some rootkit functionality: all of the sandbox's system objects are hidden from the malware binary, including modules, files, registry entries, mutual exclusion events (mutexes), and handles in general. This at least makes it much harder for the malware sample to detect the sandbox's presence. (This approach isn't undetectable, but our evaluation results show that CWSandbox generates valuable reports in practice.)

During the execution phase, heavy interprocess communication (IPC) occurs between the `cwmonitor.dll` and `cwsandbox.exe`. Each API hook function informs the sandbox via a notification object about the call and its parameters. Some hook functions require an answer from the sandbox that determines which action to take, such as whether to call the original API function. A heavy communication throughput exists because each notification object must transmit a large amount of data and several DLL instances can exist. Besides the high performance need, reliability is crucial because data must not be lost or modified on its way. Thus, we had to implement a reliable inter-process communication (IPC) mechanism with high throughput so we used a memory-mapped file with some customizations that fit our needs.

The execution phase lasts for as long as the malware executes, but the sandbox can end it prematurely when a timeout occurs or if critical conditions require instant termination of the malware.

Analysis phase

In the last phase, the sandbox analyzes the collected data and generates an XML analysis report. To measure the report's accuracy, we examined several current malware binaries and compared the results with reports generated by the Norman Sandbox and Symantec via manual code analysis.

Because the CWSandbox analyzes live systems and lets us observe how the malware binary interacts with other processes, its results were more detailed than those the Norman Sandbox provides. However, both tools generated reports that detected file system changes, registry modifications, mutex creation, or process-management actions. Only small differences between the tools exist—the reports differed if the malware binary used a random file name when it copied itself to another location, for example. Moreover, a disadvantage of Norman Sandbox is that only limited Internet connection is available; if the malware tries to download additional content from a remote location, Norman Sandbox detects it, but can't automatically analyze the remote file. In contrast, CWSandbox observes the download request and, if the malware downloads and executes a file, performs DLL injection to enable API hooking on the new process.

Compared with the reports from Symantec's manual code analysis, the sandbox reported the important actions, but it failed to detect small details and behavior variants (the creation of certain event objects, for example) because the corresponding API calls weren't hooked in the current implementation. By adding hooks to these API calls, we could extend CWSandbox's analysis capabilities. Symantec's manual code analysis didn't contain any details that weren't in our analysis report.

We executed the malware sample for a specific time period, so we used it to tune CWSandbox's throughput. We found that executing the malware for two minutes yielded the most accurate results and allowed the malware binary enough time to interact with the system, thus copying itself to another location, spawning new processes, or connecting to a remote server, and so on.

Large-scale analysis

We conducted a larger test to evaluate CWSandbox's report throughput and quality. We analyzed 6,148 malware binaries that we collected in a five-month period between June and October 2006 with nepenthes, a honeypot solution that automatically collects autonomous spreading malware.[3] Nepenthes emulates the vulnerable parts of a network's services to the extent that an automated exploit is always successful. Autonomous spreading malware such as bots and worms thus think that they've exploited the system, but rather than infecting a "victim," they're delivering to us a binary copy of the malware. Thus, our test corpus is real malware spreading in the wild; we're sure that all of these binaries are malicious because we downloaded them after successful exploitation attempts in nepenthes.

For the analysis process, we executed the sandbox on two commercial off-the-shelf systems with Intel Pentium IV processors running at 2 GHz and with 2 GBytes of RAM. Each system ran Debian Linux Testing and had two virtual machines based on VMware Server and Windows XP as guest systems. Within the virtual machines, we executed CWSandbox, effectively running four parallel environments. We stored the malware binaries in a MySQL database to which our analysis systems wrote all reports.

The antivirus engine ClamAV classified these samples as 1,572 different malware types. Most of them were different bot variants, particularly of Poebot and Padobot. Of the 6,148 samples, ClamAV classified 3,863 as malicious, most likely because signatures for the remaining binaries weren't available. The antivirus engine should have classified 100 percent of the samples as malicious, but it detected only 62.8 percent in this case.

Sample analysis report

As Figure A illustrates, CWSandbox analysis reports are quite detailed. They let analysts quickly estimate what a malware binary does and whether it needs to be further analyzed manually. With the behavior-based approach, we get a quick overview of the binary,

```
- <analysis cwsversion="Beta 1.83" time="22.11.2006 15:26:14"
  file="94c87c1c05d8f9628b789bced23f9ab3.exe"
  logpath="c:\analysis\log\94c87c1c05d8f9628b789bced23f9ab3.exe\run_1\">
  - <calltree>
  - <process_call filename="c:\94c87c1c05d8f9628b789bced23f9ab3.exe"
    starttime="00:00.046" startreason="AnalysisTarget">
  - <calltree>
    <process_call filename="C:\WINDOWS\system32\qodxih.exe C:\WINDOWS\system32\qodxih.exe
    1428 c:\94c87c1c05d8f9628b789bced23f9ab3.exe" starttime="00:05.765"
    startreason="CreateProcess"/>
    </calltree>
    </process_call>
    </calltree>
- <processes>
  - <process index="1" pid="884" filename="c:\94c87c1c05d8f9628b789bced23f9ab3.exe"
    filesize="173595" md5="94c87c1c05d8f9628b789bced23f9ab3" username="foobar"
    parentindex="0" starttime="00:00.046" terminationtime="00:06.281" startreason=
    "AnalysisTarget" terminationreason="NormalTermination" executionstatus="OK">
  + <dll_handling_section></dll_handling_section>
  + <filesystem_section></filesystem_section>
  + <mutex_section></mutex_section>
  + <registry_section></registry_section>
  + <process_section></process_section>
  + <system_info_section></system_info_section>
```

Figure A. CWSandbox analysis report. The report contains detailed information about the analyzed processes, including

CWSandbox analyzed all these binaries in roughly 67 hours: the effective throughput was more than 500 binaries per day per instance, which is at least an order of magnitude faster than human analysis. An analyst can use the resulting report as a high-level overview and analyze the binary deeper manually, if necessary.

Of the 324 binaries that tried to contact an Internet relay chat (IRC) server, 172 were unique. Because we extracted information such as the IRC channel or passwords used to access the command and control servers from the samples, we were able to mitigate the botnet risk.

Additionally, 856 of the 6,148 samples contacted HTTP servers and tried to download additional data from the Internet. By observing how the malware handled the downloaded data, we learned more about the infection stages, which ranged from downloading executable code to click fraud (automated visits to certain Web pages).

We observed 78 malware binaries that tried to use the Simple Mail-Transfer Protocol (SMTP) as a communication protocol. We recorded the destination emails and the message bodies, so we got complete information about what the malware wanted to do, which let us develop appropriate countermeasures.

More than 95 percent of the malware binary samples added registry keys to enable autostart mechanisms. Mutexes are also quite common to ensure that only one instance of the malware binary is running on a compromised host. We commonly saw malware binaries copy themselves to the Windows system folder. These patterns let us automatically define suspect behavior, and we could extend CWSandbox to automatically classify binaries as normal or malicious on the basis of observed behaviors.

We've shown that it's possible to automate binary analysis of current Win32 malware using CWSandbox. Such a tool lets analysts learn more about current malware, and the resulting analysis reports help the analyst

which is generally sufficient to extract the most important information. You can view complete sample reports at www.cwsandbox.org, as well as submit samples for analysis. The system returns analysis reports via email.

```
+ <winsock_section></winsock_section>
</process>
- <process index="2" pid="1348" filename="C:\WINDOWS\system32\qodxih.exe C:\WIN-
  DOWS\system32\qodxih.exe 1428 c:\94c87c1c05d8f9628b789bced23f9ab3.exe" filesize=
  "-1" username="foobar" parentindex="1" starttime="00:05.765"
  terminationtime="02:00.781" startreason="CreateProcess"
  terminationreason="Timeout" executionstatus="OK">
+ <dll_handling_section></dll_handling_section>
+ <filesystem_section></filesystem_section>
+ <mutex_section></mutex_section>
+ <registry_section></registry_section>
+ <process_section></process_section>
+ <system_info_section></system_info_section>
+ <window_section></window_section>
- <winsock_section>
 + <connections_unknown></connections_unknown>
 + <connections_listening></connections_listening>
 - <connections_outgoing>
 - <connection transportprotocol="TCP" remoteaddr="208.99.207.143"
   remoteport="8453" protocol="IRC" connectionestablished="1" socket="608">
 - <irc_data username="DEU|7907101" nick="DEU|7907101">
 <channel name="####test####" password="nikne" topic_deleted=":.join #a,#b,#c"/>
 </irc_data>
 </connection></connections_outgoing>
```

information about changes to the file system, the Windows registry, and the data sent via Winsock.

determine whether a manual analysis is necessary. In the future, we plan to extend CWSandbox with kernel-based hooking, which will let us monitor kernel mode rootkits and other kernel-based malware. Futhermore, we intend to explore the ways in which we can use the CWSandbox-generated reports for malware classification. ▢

References

1. I. Ivanov, "API Hooking Revealed," *The Code Project*, 2002; www.codeproject.com/system/hooksys.asp.
2. Holy Father, "Hooking Windows API—Technics of Hooking API Functions on Windows," *CodeBreakers J.*, vol. 1, no. 2, 2004; www.secure-software-engineering.com/index.php?option=com_content&task=view&id=54&Itemid=27.
3. P. Baecher et al., "The Nepenthes Platform: An Efficient Approach to Collect Malware," *Proc. 9th Int'l Symp. Recent Advances in Intrusion Detection* (RAID 06), LNCS 4219, Springer-Verlag, 2006, pp. 165–184.

Carsten Willems *is a PhD student in the Laboratory for Dependable Distributed Systems at the University of Mannheim, Germany. His research interests include malware research, including the analysis of Win32 malware. Willems has a MS in computer science from RWTH Aachen University, Germany. He is the author of CWSandbox, a tool for automatic behavior analysis. His company, CWSE GmbH deals with software development in IT security. Contact him at cwillems@consolo.de.*

Thorsten Holz *is a PhD student in the Laboratory for Dependable Distributed Systems at the University of Mannheim, Germany. His research interests include honeypots, botnets, malware, and intrusion detection systems. Holz has an MS in computer science from RWTH Aachen University, Germany. Contact him at thorsten.holz@informatik.uni-mannheim.de.*

Felix Freiling *is a professor of computer science and heads the Laboratory for Dependable Distributed Systems at the University of Mannheim, Germany. His research interests include the theory and practice of dependability. Freiling has a PhD in computer science from Darmstadt University of Technology, Germany. Contact him at freiling@informatik.uni-mannheim.de.*