Malyzer: Defeating Anti-detection for Application-Level Malware Analysis

Lei Liu and Songqing Chen

Department of Computer Science
George Mason University
{lliu3,sqchen}@cs.gmu.edu

Abstract. Malware analysis is critical for malware detection and prevention. To defeat malware analysis and detection, today malware commonly adopts various sophisticated anti-detection techniques, such as performing debugger, emulator, and virtual machine fingerprinting, and camouflaging its traffic as normal legitimate traffic. These mechanisms produce more and more stealthy malware that greatly challenges existing malware analysis schemes.

In this work, targeting application level stealthy malware, we propose Malyzer, *the key of which is to defeat malware anti-detection mechanisms at startup and runtime so that malware behavior during execution can be accurately captured and distinguished.* For analysis, Malyzer always starts a copy, referred to as a shadow process, of any suspicious process on the same host by defeating all startup anti-detection mechanisms employed in the process. To defeat internal runtime anti-detection attempts, Malyzer further makes this shadow process mutually invisible to the original suspicious process. To defeat external anti-detection attempts, Malyzer makes as if the shadow process runs on a different machine to the outside. Since ultimately malware will conduct local information harvesting or dispersion, Malyzer constantly monitors the shadow process's behavior and adopts a hybrid scheme for its behavior analysis. In our experiments, Malyzer can accurately detect all malware samples that employ various anti-detection techniques.

1 Introduction

Internet malware poses an immense threat to computer system security. Fundamentally, malware aims to collect local sensitive information, such as bank account information, password, and CD keys, or leverage infected hosts for various attacks, such as spam relay, DDoS, IP laundering (acting as stepping stones), and phishing. These malicious actions are commonly referred to as information harvesting and information dispersion [15], respectively.

A number of schemes have been proposed and used for malware detection and analysis. Among them, the signature-based approach has been employed for many years and is the most prevalent scheme in practice. In general, signature-based schemes [8,19,20,28] generate content-based signatures that can uniquely identify the malware. Signature-based schemes are efficient and effective in detecting and containing known malware, but they are inherently ineffective against

previously unknown or polymorphic/metamorphic malware [23]. Encryption [10] and obfuscation [25] are also commonly used to make the signature-based schemes incompetent.

Research has been conducted on behavior analysis that complements the content-based signature approach. Behavior analysis aims to identify abnormal process behavior. In general, the process behavior under supervision is compared with a pre-defined safe model. Any behavior deviating from the predefined model would lead to an alarm, which is in line with the anomaly-based approach. For behavior analysis, various approaches have been studied, such as using dynamic taint processing [11,12,24,29,31], checking auto-start extensibility points in registry [30], and searching for various hooks [9,27].

However, due to underlying economic motivations, various anti-detection mechanisms have been constantly and continuously developed and quickly adopted by malware developers to make more and more stealthy malware. The efforts are from two perspectives. One is to use protective camouflaging to conceal its existence. For example, modern bots try to blend their traffic with normal user traffic [16,21]. Some malware [1] may run as browser helper objects (BHO). Advanced malware [2] is found to perform dynamic code replacement. That is, a benign user application process is started, and malware code is then written to its memory sections and executed in the context of this process. When misbehavior is identified, it will be traced down to a "legitimate" application process. This approach becomes more and more common since malware can easily go through firewalls via such an approach.

Besides camouflaging at runtime, malware developers today also commonly adopt proactive approaches to evade detection at startup. For example, as malware analysis and detection may use various debugger, emulator, or run suspicious code samples in a virtual machine environment, most of today's malware performs virtual machine, emulator, and debugger detection, which we refer to as running environment tests, before the logic of malware gets executed [3].[1] These techniques make systems like Panorama [31] (that relies on an emulator for malware analysis) and SpyProxy [22] (that executes the Web content in a virtual machine) to stop functioning. Widely adopting these anti-detection techniques in malware, malware developers successfully enforce malware analysis and detection to be conducted in a real executing environment, in which various runtime camouflaging as aforementioned can effectively protect malware.

Targeting application-level stealthy malware, in this paper, we propose Malyzer, an execution-based approach for malware analysis. *The key idea of Malyzer is to unveil malware camouflaging at startup and runtime so that malware behavior can be accurately captured and distinguished.* For this purpose, Malyzer constantly monitors processes' startup procedures. If a process is suspicious (i.e., analysis object), Malyzer thus can start a copy of this process, which is referred to as a shadow process, on the same host no matter what anti-detection techniques have been employed at startup. Malyzer makes the shadow process inaccessible

---

[1] Such tests could be combined into a packer that can perform multi-layer packing for anti-reverse-engineering [17] against static malware code analysis.

to other processes or users in order to eliminate noisy activities that do not belong to the shadow process. Without interferences caused by other processes or users on the same host, the behavior of the shadow process could only be autonomous by its inner logic or caused by some remote control.

Because the shadow process runs in the same environment as the original suspicious process, the environment test including virtual machine detection , should that be performed, will always pass. To defeat internal anti-detection tests, Malyzer further makes this shadow process mutually invisible to the original process. Malyzer also controls direct user accesses to this shadow process in order to minimize legitimate user activities that malware could leverage for camouflaging. To defeat external anti-detection attempts, Malyzer makes the shadow process running as if it is running on a different machine to the outsider. Since ultimately malware will conduct local information collection or dispersion, Malyzer constantly monitors the shadow process's disk, network, and memory accesses and uses a hybrid scheme combing both anomaly-based and signature-based approaches for process behavior analysis. We have implemented a prototype system of Malyzer. Our evaluation results show that it can accurately capture the behavior of all the malware samples that use various anti-detection techniques in our experiments.

The rest of the paper is organized as follows. We present Malyzer design in section 2 and prototype implementation in section 3. Malyzer is evaluated in section 4. We further discuss some optimizations and limitations of Malyzer in section 5 and make concluding remarks in section 6.

2   Malyzer Design

Figure 1 shows the system architecture of Malyzer design. Malyzer consists of three components: `Startup Tracker`, `Shadow Process Manager`, and `Shadow Process Monitor`.
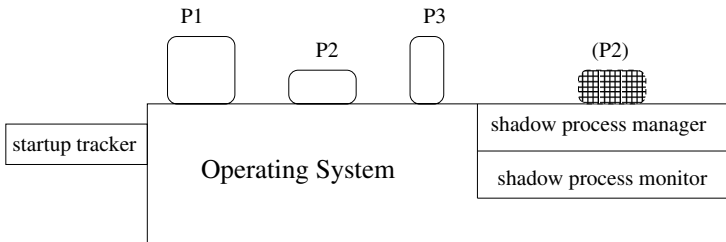

Fig. 1. System Architecture. Suppose the process P2 is suspicious. Through `Startup Tracker`, information regarding how P2 is started is fetched as well as a memory image of P2 (if necessary). No matter how P2 is started, a shadow process of P2 could be started on the same host by `Shadow Process Manager`, which defeats all kinds of anti-detections. Then `Shadow Process Monitor` monitors the disk/memory/network accesses of the shadow process for malware behavior analysis.

2.1   Startup Tracker

To start a shadow process of the given suspicious process, Malyzer must obtain the runnable executable of the malware so that a shadow process can be started on demand. With sophisticated anti-detection mechanisms, such as dynamic code replacement, however, obtaining the executable of a given process is sometimes non-trivial. For example, directly dumping the memory image of the given process often does not work.

   In general, today malware runs in three possible forms and there are three types of relationships between a malware process and the corresponding executable.

- Malware runs directly from the executable (possibly with multi-layer packing) on the disk. This is the most common and trivial approach. The malware process starts when the system starts (usually it is achieved by modifying registry entries). Given a process, it is easy to locate its executable on the disk by querying the module name, through which the full path of the executable can be obtained. Many traditional virus and worms use this approach.
- Malware runs as a DLL in the context of a benign application process. In this approach, the malware is encapsulated into a DLL, and then installed statically to a benign application or dynamically to a running process. For example, when a user browses a particular Web site, the user is fooled to install a helper object to the Web browser. `WebBuying` [1] uses this approach.
- Malware runs through dynamic code replacement. This is an advanced and prevailing approach. The common thread of this approach is as follows:
   1. The malware first starts a benign process in a suspended mode. The system API `createProcess`[2] is normally used to invoke the benign process.
   2. The malware then allocates memory in the domain of the suspended process and injects its own executable. This is often conducted through `writeProcessMemory`.
   3. The malware then sets entry point and resumes execution of suspended process by using `resumeThread` or `createRemoteThread`. The logic of benign process is completely skipped.

   Figure 2 shows an example of dynamic code replacement of graybird, which is one of the most prolific piece of Windows malware [2]. With wide usage of this approach, querying the module name only returns the genuine executable of a benign process on the disk, if an identified suspicious process is given. This not only allows the malware to go through firewalls which is a crucial design target of modern malware, but also misleads malware analysis and detection.

   With these three approaches and the commonly adopted other anti-detection mechanisms, it is difficult 1) to locate the runnable malware executable; 2) even if a malware code sample is identified, it may not be start-able.

---

[2] A family of APIs can be used for process creation, such as `createProcessAsUser` and `createProcessWithLogonW`. We use `createProcess` to represent all of these. We take a similar approach for other APIs for brevity.
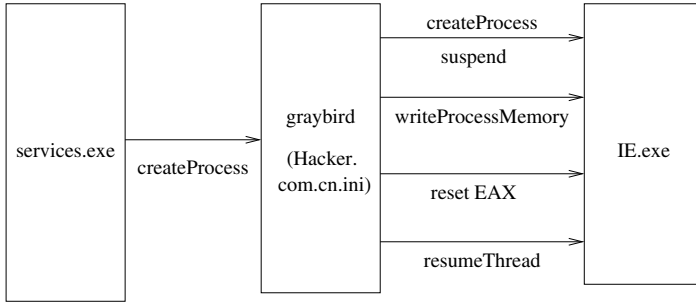
Fig. 2. Dynamic Code Replacement of `graybird`. `graybird` camouflages itself as Internet Explorer (IE). It starts an IE process, and then replaces the memory image of IE with graybird. After resetting EAX, it resumes the process to execute the real malcode of graybird, but appears to be an IE process in the system.

Dynamic code replacement/injection is not uncommon in normal programming practice. With the increasing use of encryption technologies and packers, dynamic code replacement has been used by a lot of benign applications for all kinds of purposes, such as code protection [14]. But we noticed that dynamic code replacement adopted by benign applications is usually confined within the application processes, and it's rare that dynamic code replacement requires to start another process in these benign applications.

Therefore, `Startup Tracker` is designed to differentiate these situations and to preserve critical startup information for `Shadow Process Manager`. In addition, `Startup Tracker` can also provide hints for malware detection. For example, a benign process would rarely start with a **inter-process** dynamic code replacement approach.

Running as a driver to the host operating system, `Startup Tracker` tracks the process creation procedure once the machine is started. In Windows systems, Windows does not provide a convenient mechanism to track process creation and termination. Malyzer thus keeps listening to all process creation notification messages. A notification message typically includes information such as a process ID and a parent process ID. In Unix-like systems, such information can be easily obtained from the process data structure.

To identify different process startup approaches, `Startup Tracker` also needs API level information to discover possible dynamic code replacement. `Startup Tracker` achieves this through identifying a unique API call sequence. That is, as aforementioned, for dynamic code replacement, the successive system API calls of `createProcess`, `writeProcessMemory`, and `ResumeThread` are inevitable. Thus, when `Startup Tracker` monitors the process startup procedure, if a sequence of the above system API calls with appropriate parameters is identified, it is highly likely to be dynamic code replacement, which is suspicious.

In addition, `Startup Tracker` also needs to prepare for starting a shadow process of any identified suspicious process as we discuss in the next section. Therefore, if dynamic code replacement is found, `Startup Tracker` also dumps

the process initial memory image before the process is resumed as well as recording other information, such as section sizes and the entry point.

If a malware process runs directly from the executable on the disk or runs as a DLL of a benign application, `Startup Tracker` can provide the child-parent information regarding how a process is started. If the malware runs as a DLL to a benign application, we will discuss, in the next section, how to match up the DLL(s) in the shadow process at runtime.

2.2   Shadow Process Manager

The role of `Shadow Process Manager` is mainly to defeat runtime malware anti-detection mechanisms that come from the internal or external sources. Thus, `Shadow Process Manager` needs to make the shadow process to run as a normal process, which is mutually invisible to the original process to deal with internal anti-detection mechanisms, while it is accessible to the outside to deal with any external anti-detections.

2.2.1   Defeating Internal Anti-detections: The Shadow Process Is Mutually Invisible to the Original Process

With the help of `Startup Tracker` that has done sufficient preparation, `Shadow Process Manager` can start the shadow process with ease. However, `Shadow Process Manager` also needs to guarantee a normal status of the shadow process at runtime since various techniques may be used by a malware instance to perform anti-detections during its execution. For example, a simple but commonly used detection performed by a malware process is to frequently check if there are multiple instances of itself running on the same host. If there are, the process terminates. In our experiments, nearly all malware samples adopt some mechanisms to prevent multiple instances from running on the same host. To defeat these anti-detection mechanisms, `Shadow Process Manager` thus needs to use an uncommon approach to start the shadow process, and hook up some interceptors when the shadow process is started.

To detect multiple instances, usually a process could use the following mechanisms:

- **through shared memory section:** Statement like
  $\#pragma\ comment(linker, ``/section : SHARED, RWS'')$ can set up a shared section between different instances of the same process. A process can access the section directly and detect if it has been modified by other process instance. It is also possible to create a shared section with API like `NtCreateSection` or `CreateFileMapping` at run time. Different process instances can only access the shared section through a name or file handle. So we treat this case as a named object.
- **through process list checking:** The process can emulate the process list and check if there is any process with the same module name as itself. This approach is not used if the malware runs through DLL injection.

– **through named object:** Named objects provide a convenient approach for processes to share object handles. After a process has created a named event, mutex, semaphore, file-mapping, section or timer object, other processes can use the name to open the handle to the object. If a process tries to create an object using a name that is in use by another process, the function fails and `GetLastError` returns `ERROR_INVALID_HANDLE`. In general, mutex is the most commonly used named object.

Among these mechanisms, accessing shared memory section cannot be intercepted. If a memory section is shared, the section has the corresponding `SHARED` property. When the operating system (OS) loads this section, the OS checks whether the section exists. If it does, the OS only points the section to the existing one so that multiple process instances will access the same memory region.

One possible solution to deal with this in the shadow process (in order to prevent possible multiple copy detection) is to copy the portable executable (PE) to a file with a different name. The solution is simple and effective but it also has limitations. In our experiments, all malware samples query the module name, and proceed according to the query result. Although it is possible to add additional interceptors to change the module name, it introduces new complexity.

Thus, `Shadow Process Manager` takes another approach, similar to dynamic code replacement used by malware, in which the PE content is read, aligned, and copied to a suspended process. In this procedure, however, the copy does not honor the `SHARED` property of memory sections. It simply allocates new memory space for all sections and then copies everything. Thus, the shared section would present as a new section in the memory [18]. The suspended process, which we refer to as a shell, could be any process with respect to defeating the memory sharing approach.

Considering the possible usage of shared memory section, `Shadow Process Manager` thus always starts the shadow process using a shell. Since the original suspicious process is running in the system, its shell PE must exist. Malyzer can always start a shell from this PE with the right module name. Then

– If no dynamic code replacement is found by `Startup Tracker`, `Shadow Process Manager` will launch the shadow process based on the portable executable (PE) on the disk. That is, `Startup Process Manager` reads the PE, aligns, and writes to the memory.
– If dynamic code replacement is found, `Shadow Process Manager` will start the shadow process based on the dumped memory image by `Startup Tracker`. That is, `Shadow Process Manager` will create the shell, then copy the memory image into the process and resume the execution.
– In either of the above two situations, some malcode may be injected as a DLL into the process dynamically. `Shadow Process Manager` needs to guarantee that the shadow process load all DLLs as the original suspicious process. Therefore, after a shadow process is launched, Malyzer further compares the DLL list of the shadow process with that in the original suspicious process. If any DLL is missing in the shadow process, `Shadow Process Manager` will insert the corresponding DLL into the shadow process.
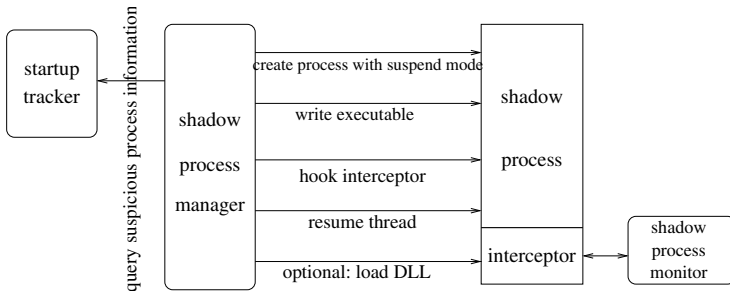
Fig. 3. Shadow Process Manager. A shadow process is started. The corresponding executable could be the dumped image or PE on the disk.

Figure 3 illustrates how `Shadow Process Manager` works to start a shadow process and interfaces with `Startup Tracker` and `Shadow Process Monitor`.

By this way, upon the shadow process startup, `Shadow Process Manager` has already defeated possible usage of shared memory sections in the malware. To deal with the malware anti-detection through process list checking and named objects, `Shadow Process Manager` further intercepts system API calls from the shadow process after the shadow process is started, and

 – If the process checks the process list, `Shadow Process Manager` can make the original process invisible to the shadow process. In general, process list checking is through `CreateToolhelp32Snapshot`, which returns a snapshot of all processes, and then `Process32Next` is commonly used to emulate all processes. Thus, `Shadow Process Manager` only needs to intercept `Process32Next`. When `Process32Next` is ready to return the original suspicious process, it is skipped and the next process on the list is returned.
 – If the process checks named objects, `Shadow Process Manager` can address this through system API interception. In general, a named object is accessed through a unique name. For example, if a mutex is used, `CreateMutex` takes the name of the mutex as a parameter. `Shadow Process Manager` generates two independent name domains for the original and shadow processes. The interceptor thus can replace the name parameter of the shadow process with a different one to avoid name conflict. The interceptor also returns the original name if the shadow process queries the name of the named object.

Apparently, the above is regarding how to make the original suspicious process invisible to the shadow process. In order to make the shadow process invisible to the original suspicious process, only `Process32Next` needs to be intercepted in the original process.

To facilitate the malware behavior analysis in the `Shadow Process Monitor`, Malyzer also makes the shadow process inaccessible to direct user accesses in order to reduce the analysis noise. For this purpose, Malyzer intercepts `ShowWindow` and `ShowWindowAsync`, always replaces parameter `nCmdShow` with `SW_HIDE`. The shadow process is then inaccessible to direct user input and misbehavior of the shadow process can be accurately captured.

2.2.2   Defeating External Anti-detections: The Shadow Process Behaves Normally to the Outside

Now the shadow process can run in parallel to the original suspicious process. But it may still not behave "correctly" as a normal malware instance.

It is common that malware instances need to interact with some outsider. For example, a bot master controls all bots, and bots running on individual hosts must communicate with the bot master in order to receive commands, send back collected local information, etc. Thus, the shadow process must have the networking capability, and can communicate with outside if needed.

However, in setting up the correct networking functionalities of the shadow process, the outsider, such as a bot master, may have restrictions. For example, multiple connection requests from the same IP address may not be allowed on an IRC server. Thus, it is important that the shadow process should present as another malware instance on a different host to the outsider.

Even without such constraints from the outsider, the malware instance may always use a pre-determined port to communicate with the outside. Once the original process is bound to a particular port, the shadow process cannot use that port, which may cause the malware instance to stop functioning.

Thus, Malyzer must bind the shadow process to a different IP address from the original suspicious process, but use the same port number so that an outsider cannot figure out they are from the same host. That is, Malyzer must be able to support multiple IP addresses.

Usually a process calls `connect` to connect to a remote IP address and calls `bind` to bind to a specific port before it begins to accept connections. The default IP address for these APIs is the primary IP address. In order to bind the shadow process to a secondary IP address, Malyzer needs to intercept these API calls and add an additional IP address in the OS for this purpose.

In Malyzer, when `connect` is called, the socket is always bound to the primary IP address (`A.B.C.D1`) by default. In order to have the shadow process bound to the secondary IP address, Malyzer actively performs binding to (`A.B.C.D2`) before `connect` is called in the API interceptor. When receiving incoming connections, Malyzer intercepts `bind` to bind port to (`A.B.C.D2`) to avoid port conflict. These networking setups guarantee that the shadow process appear to the outsider as a new process running on a different machine.

2.3   Shadow Process Monitor

Malware always performs local sensitive information collection, or controls infected nodes to participate some attacks against a third party. This is fundamental to differentiate a benign process from a malware process. Thus, after the shadow process successfully runs, Malyzer keeps monitoring the behavior of the shadow process in order to determine if it is actually a malware instance.

Considering that ultimately, a malware instance would perform information harvesting or information dispersion, Malyzer concerns three typical kinds of behavior observed from the shadow process:

- **network accesses:** A lot of malware has network activities for various purposes. For example, a trojan sends out the information it collects; a bot contacts its botmaster for commands.
- **hard disk access:** Some malware is designed to collect sensitive information stored on the hard disk of infected hosts. In general, it is normal that a benign process accesses certain directories on the hard disk, such as the current working directory of the process, but it is uncommon to access directories owned by other processes.
- **memory access:** Some malware can directly access memory of other processes for sensitive information (e.g., password) or for further attacks.

In order to decide whether or not a process with a sequence of accesses is malware, existing research commonly applies either anomaly-based analysis or signature-based schemes. But the challenge is that application (including malware) behavior is difficult to predict, considering the complexity of software and diversity of user operations. Fortunately, in Malyzer, the situation is different. The shadow process is a clone of the original suspicious process. Malyzer makes it inaccessible to other processes or users in the host system. This greatly eliminates noisy activities that do not belong to the shadow process (we discuss how to deal with interactive malware with emulated user input later). Without interferences caused by other processes or users on the same host, the behavior of the shadow process could only be autonomous by its inner logic or caused by some remote control. Even though, simply using an anomaly-based approach may not be easy since it is arduous to *manually* define a normal behavior model for various shadow processes of legitimate application processes.

At this end, `Shadow Process Monitor` takes a hybrid approach: to combine both anomaly- and signature-based approaches. Malyzer first defines a set of heuristic malicious behavior rules and then generates the normal process behavior model automatically along with the malware analysis. Malyzer always starts with the anomaly-based approach (the benign process behavior model, consisting of individual process profiles, is empty at the beginning). Malyzer first compares the shadow process behavior with the benign process model. If the model is empty or it cannot make a decision, it is further compared against the heuristic rules. Once the process is determined to be benign, its profile is generated automatically from the captured access sequence and added to the benign process behavior model. Note that an optional component, the user validation, can be added when `Shadow Process Monitor` determines a malware instance. This could improve the accuracy of the analysis. Without such a component, the procedure is fully *automatic*.

Considering three types of accesses, Malyzer defines the heuristic rule set for malicious behavior detection of the shadow process as follows:

- **connection rate (rule I):** When connection count to different destinations in a given time duration is beyond a threshold, an alarm is raised.
- **failed connection rate (rule II):** When the number of failed connections a shadow process makes to different destinations in a given time duration is beyond a threshold, an alarm is raised.

- **command and control channel (rule III):** When the shadow process maintains a certain number of connections beyond a threshold or when it maintains a connection to a destination for a duration beyond a threshold, an alarm is raised.
- **sensitive file on disk (rule IV):** When the shadow process accesses directories other than system directory and current working directory, an alarm is raised.
- **sensitive data in memory (rule V):** When the shadow process reads or writes the memory of other processes, an alarm is raised.

Combination of these heuristic rules could be applied too. Note that these heuristic rules are effective in Malyzer because a significant amount of noise caused by users has been eliminated.

On the other hand, for each process, the normal behavior model consists of the process profile list, a list of relevant API calls and corresponding parameters when network, disk, and memory accesses are invoked in the shadow process.

3   Malyzer Implementation

To demonstrate the concept and for experiments, we implement a prototype of Malyzer on Windows XP Professional. As we have shown in Figure 3, the three major components of Malyzer interact with each other. Among them, `Startup Tracker` is implemented as a driver to keep track of process creation. It keeps monitoring and intercepting system daemons, such as `services.exe`, `svchost.exe`, `lsass.exe`, `spoolsv.exe`, and `system.exe`. This is done through directly replacing their import tables entries [26]. In addition, `Startup Tracker` keeps intercepting process creation related APIs, such as `createProcess`, `writeProcessMemory`, and `resumeThread`, which is done through Microsoft `Detours 2.1 Express` [4]. Upon a child process is created by an intercepted process, relevant APIs in child process are also intercepted in a recursive fashion.

If `Startup Tracker` detects dynamic code replacement via the unique API call sequence as we have discussed in section 2, `Startup Tracker` will dump the initial memory of the process. Memory dumping is conducted before the process is actually resumed. In addition, the EAX register containing the entry point address, obtained from the parameter of `resumeThread`, is kept. `Startup Tracker` stores this information for `Shadow Process Manager`.

`Shadow Process Manager` is responsible for starting a shadow process on demand, given a suspicious process to analyze. `Shadow Process Manager` first queries the module file name of the suspicious process and creates a process with suspended mode from the corresponding executable.

By querying `Startup Tracker`, if the suspicious process is not started via dynamic code replacement, `Shadow Process Manager` reads the executable again, aligns, and copies that to the suspended process again (in order to defeat direct memory sharing in malware instances). This procedure is the same as dynamic code replacement. In addition, `Shadow Process Manager` also calculates the entry point address and sets EAX for the shadow process [18].

If the suspicious process is started via dynamic code replacement, `Shadow Process Manager` reads the memory image dumped by `Startup Tracker` and copies to the suspended process. As `Startup Tracker` also keeps the EAX value of the suspicious process, `Shadow Process Manager` can calculate the entry point from the original EAX value and set accordingly in the shadow process.

Before resuming the shadow process, `Shadow Process Manager` hooks interceptors to the shadow process in order to detect various anti-detection approaches that could be used by a malware instance and monitor its behavior.

Some of the intercepted API calls and their parameters from `Shadow Process Manager` are also fed into `Shadow Process Monitor` for further decisions. These are disk, network, and memory accessing APIs and their parameters. Such a call is an event to trigger `Shadow Process Monitor`. In the current implementation, `Shadow Process Monitor` maintains a cache, which contains shadow process profiles. Once an API calling event is fed to `Shadow Process Monitor`, it compares with its cached profiles. Currently, the API name and the corresponding parameters are compared. A discrepancy will raise an alarm.

For heuristic rules, we have set up thresholds as follows: the connection rate and the failed connection rate are set as ten and five per minute. For the command and control channel, the connection duration threshold for a persistent channel is 30 seconds [7]. If the shadow process is finally determined to be benign, its API calls are recorded and updated in the cache.

In addition to capture behavior of malware shadow process, Malyzer also always locates the correct malcode source. The difficulty lies in that if the malware runs via DLL injection, such as BHO, Malyzer is expected to report which DLL the API caller is from. In our current implementation, Malyzer queries stack information [13] to find the caller upon an API call triggering an alarm. For this purpose, Malyzer defines a macro that reads register `EBP`, which contains the value of frame pointer of the caller. Subsequently, the return address is stored at (`EBP + 4`). With the return address, Malyzer can query the module where the return address resides.

4   Malyzer Evaluation

With the prototype, we test Malyzer against a number of malware samples that use different anti-detection mechanisms. Some representative experimental samples that use different anti-detections are listed in Table 1. We experiment on all malware samples. Due to page limit, we will only present some interesting ones with different anti-detection mechanisms.

4.1   Whether Malyzer Can Defeat Malware Anti-detections

We first test whether a shadow process of them can be successfully started on the same host where there is already one malware instance running.

`reptile` is the most interesting sample we tested. In this version, `reptile` performs various environment tests to defeat virtual machines and various

Table 1. Malware Samples and their Anti-detections

| malware	anti-detection actions
agobot3	check process list for processes with the same name perform VMware detection
forBot	check process list for processes with the same name
graybird	start with dynamic code replacement
Gorgon trojan	check process list for debugger process OLLYDBG.EXE
JrBot	use mutex to prevent multiple copies from running on a host
reptile	perform debugger, VMware, SoftIce detection perform BreakPoint, Single Step detection use mutex to prevent multiple copies from running on a host
rBot	use mutex to prevent multiple copies from running on a host
sdbot05	check process list for processes with the same name
spybot	perform VMware detection
storm worm	perform VMware detection
trojan downloader	Search for Wireshark, ZoneAlarm, Olly Debug
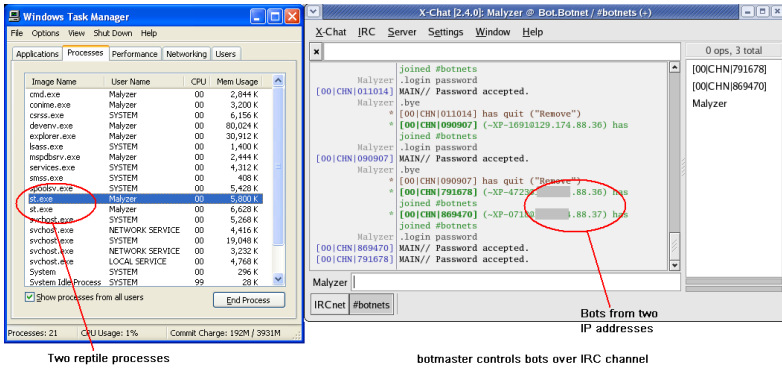
Fig. 4. A shadow process of `reptile` is started and running on the same host

debuggers and debugging techniques. It also prevents multiple copies from running on the same host via mutex. When `reptile.exe` starts, it copies itself to `%windir%\st.exe` and launches `st.exe` before it exits. Subsequently, `st.exe` deletes `reptile.exe` on the disk. After `Shadow Process Manager` starts a shadow process of this malware instance, Figure 4 shows that two processes are running successfully within Malyzer. `Graybird` runs via dynamic code replacement so that it can disguise as an IE process. `Startup Tracker` detects this approach and dumps the initial process memory. Accordingly, `Shadow Process Manager` starts a shadow process successfully. Figure 5 shows that two graybird processes are running in the context of two IE processes. `agobot3` is a very common bot that employs the process list checking to prevent multiple copies from running on the same machine. Figure 6 shows that in Malyzer, a shadow process of `agobot3` can be started and run in parallel with the original one successfully.
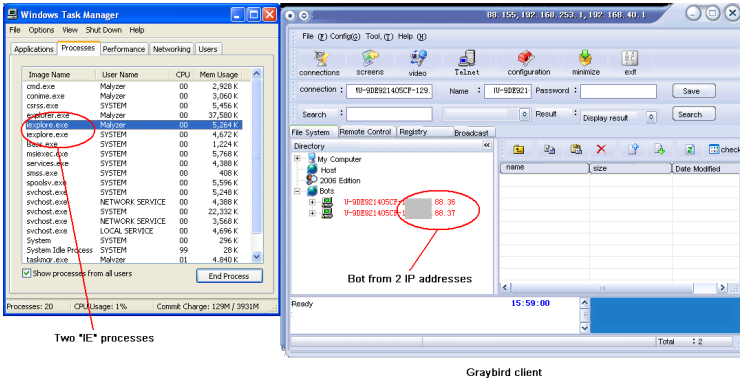
Fig. 5. A shadow process of `graybird` is started and running on the same host. `graybird` starts itself via dynamic code replacement.
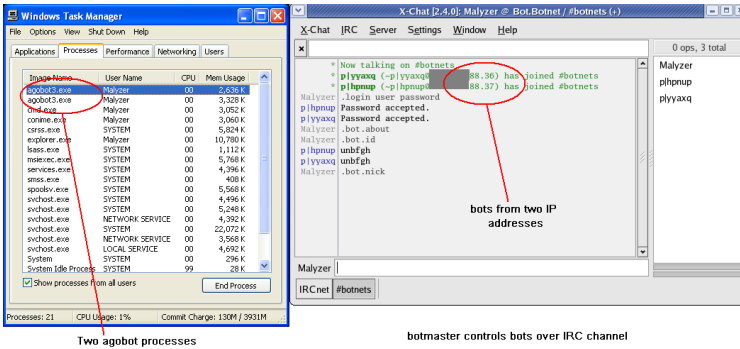

Fig. 6. A shadow process of `agobot3` is started and running on the same host. `agobot3` simply checks the process list to prevent duplicated copies from the same IP address.

In these tests, both `agobot` and `reptile` are IRC-based malware. They need to connect to an IRC server to receive commands. By default, the IRC server only accepts one connection per IP address. Because Malyzer binds the original process and the shadow process to a different IP address, although they are running on the same host, the IRC server takes them as two different instances running on two different machines, as shown in the above figures.

4.2 Whether a Shadow Process Functions Normally and Whether Its Misbehavior Can Be Detected

The previous tests show that the shadow process of a malware sample can be successfully started. However, whether the shadow process can still function "properly" is not clear, which is critical for the further detection and analysis.

Table 2. Detection of Information Harvesting and Dispersion by rBot

action	event trigger	detection rule
After the process starts, it connects to the bot master.	`connect`: shadow process keeps a consistent connection	rule III: command and control channel
After rBot receives a `advscan` command, it starts random port scan.	`connect`: shadow process sends out packets to different destinations, the connection rate is beyond a threshold.	rule I and rule II: connection rate and failed connection rate
After rBot receives a `getcdkeys` command, it retrieves cdkey file.	`ReadFile`: shadow process accesses file of other applications.	rule IV: sensitive data access on disk

To verify that a shadow process can work "correctly" as if it is truly running on a different host, we experiment by instructing the shadow process for various attacks. Along with these experiments, we can also test whether their misbehavior can be accurately captured so that a decision is made through `Shadow Process Monitor`. At the beginning, the cache in `Shadow Process Monitor` is empty, and the heuristic rules are used. Table 2 shows the results when `rBot` is instructed to perform local and remote attacks. For local attacks, a command of `getcdkeys` from our IRC server is sent to the bot to search for product CD keys. For remote attacks, a command of `advscan` is sent to instruct the bot to perform random port scan. In the experiments, all malware actions are correctly captured by Malyzer.

5   Malyzer Optimization and Further Discussion

In this section, we discuss some issues with the current design and implementation of Malyzer as well as possible improvement.

First, our experiments tested automatic malware. For malware whose misbehavior is triggered by certain user activities, such as accessing a specific Web site, the current Malyzer implementation has not included a component to use emulated user input to allure malware actions. We are adopting the approach taken by Panorama [31] to emulate the user input to trigger the malware.

Second, in the design space, `Shadow Process Monitor` aims to capture misbehavior of shadow processes. This approach is similar to existing behavior-based approaches with the understanding that information harvesting and information dispersion are the essential behaviors differentiating malware processes from benign ones. Other approaches and systems (e.g., Snort, Bro) could be integrated with this component. In addition, the current cache implementation of `Shadow Process Monitor` is rather simple for the demonstration of concept. We would like to define a general and portable format so that once a shadow process profile is created, it can be shared and distributed.

Third, the current implementation of Malyzer relies on behavior analysis through API call interceptors. A malware developer could evade Malyzer without calling such APIs by coding its own functions or calling some system native functions that we are not aware of to evade Malyzer. Although Malyzer can be implemented at the native system call level to defeat these efforts, fundamentally, a malware developer can defeat Malyzer by implementing in assembly code or using timing correlation to detect the existence of Malyzer by looking into the time duration of calling various APIs. Although our usage of Detours causes trivial processing overhead as reflected by our successful experiments with `reptitle`, which uses API timing to defeat "Single Step", more precise timing could enforce Malyzer to intercept more APIs in order to defeat such efforts.

On the other hand, as a malware analyzer, Malyzer works under the assumption that the host is not subverted through some rootkits. Whether at the stage to unveil malware camouflaging or de-activate various malware anti-detection mechanisms, Malyzer needs a trustworthy underlying operating system. Once the underlying operating system is completely subverted, the information Malyzer obtains could be wrong, which would lead to analysis failure.

In addition, for duplicated process checking, a malware instance could mark the registry or a file so that it can query the mark at the startup. While Malyzer cannot defeat this, such an anti-detection approach is not reliable because if exceptions happen (e.g., powering down), the malware cannot restart.

Lastly, today a lot of malware packers are available for malcode polymorphism, obfuscation, encryption, and some anti-detection as well. For example, Themida [5] provides anti-debug, anti-tracing, anti-dumping and VM detection options. While source polymorphism, obfuscation, and encryption do not affect Malyzer, since Malyzer uses Detours to intercept API calls, which intercepts Win32 functions by re-writing target function code in memory, some packers can use IAT destruction to completely change the PE structure of malcode. In this case, Detours fails to intercept API calls. As an alternative, proxy DLL [6] does not rely on the IAT structure which is a good candidate of the API interceptor. Malyzer thus can take this approach.

6   Conclusion

Various anti-detection techniques have been practically employed by malware developers to create more and more stealthy Internet malware. The success of malware analysis and detection heavily depends on whether malware analysts can defeat all kinds of malware anti-detection mechanisms. In this paper, we have made an initial step towards effectively unveiling various malware camouflaging at startup and runtime through the design and implementation of Malyzer. Through various countering anti-detection measures, Malyzer can accurately capture malware behavior. Experiments have been conducted to evaluate Malyzer with various malware samples. The results demonstrate the effectiveness of Malyzer.

Acknowledgment

References

1. http://www.pctools.com/mrc/infections/id/Webbuying/
2. http://news.softpedia.com/newsTag/Graybird
3. http://blogs.windowsecurity.com/parker/2006/07/11/malware-packers/
4. http://research.microsoft.com/sn/detours/
5. http://www.oreans.com/ThemidaWhatsNew.php
6. http://www.codeproject.com/KB/system/hooksys.aspx
7. Taxonomy of botnet threats (November 2006),
   http://us.trendmicro.com/imperia/md/content/us/pdf/threats/
   securitylibrary/botnettaxonomywhitepapernovember2006.pdf
8. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proceedings of IEEE Symposium on Security and Privacy, Berkely/Oakland, CA (May 2006)
9. Butler, J., Hoglund, G.: Vice-catch the hookers! (July 2004)
10. Chiang, K., Lloyd, L.: A case study of the rustock rootkit and spam bot. In: Proceedings of the First Workshop on Hot Topics in Understanding Botnets, Cambridge, MA (April 2007)
11. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (August 2004)
12. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: Proceedings of SOSP, Brighton, United Kingdom (October 2005)
13. Dimitrov, C.: Playing with the stack,
    http://www.codeproject.com/tips/stackdumper.asp
14. Desclaux Fabrice. Skype uncovered, http://www.ossir.org/windows/supports/
    2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf
15. Grizzard, J., Sharma, V., Nunnery, C., Kang, B., Dagon, D.: Peer-to-peer botnets: Overview and case study. In: Proceedings of the HotBots, Cambridge, MA (April 2007)
16. Gu, G., Zhang, J., Lee, W.: Botsniffer: Detecting botnet command and control channels in network traffic. In: Proceedings of the 15th NDSS, San Diego, CA (February 2008)
17. Kang, M., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of WORM, Alexandria, VA (November 2007)
18. Keong, T.: Dynamic forking of win32 exe,
    http://www.security.org.sg/code/loadexe.html
19. Kim, H., Karp, B.: Autograph: Toward automated distributed worm signature detection. In: Proceedings of USENIX Security, San Diego, CA (August 2004)
20. Li, Z., Sanghi, M., Chen, Y., Kao, M., Chavez, B.: Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In: Proceedings of IEEE Symposium on Security and Privacy, Berkely/Oakland, CA (May 2006)

21. Liu, L., Chen, S., Yan, G., Zhang, Z.: Bottracer: Execution-based bot-like malware detection. In: Proceedings of the 11th Information Security Conference, Taipei, China (September 2008)
22. Moshchuk, A., Bragin, T., Deville, D., Gribble, S., Levy, H.: Spyproxy: Execution-based detection of malicious web content. In: Proceedings of the 16th USENIX Security Symposium, Boston, MA (August 2007)
23. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA (May 2005)
24. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12th NDSS (February 2005)
25. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The ghost in the browser analysis of web-based malware. In: Proceedings of the First Workshop on Hot Topics in Understanding Botnets, Cambridge, MA (April 2007)
26. Richter, J.: Programming applications for microsoft windows
27. Rutkowaska, J.: System virginity verifier: Defining the roadmap for malware detection on windows systems (September 2005)
28. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Proceedings of OSDI, San Francisco, CA (2004)
29. Stinson, E., Mitchell, J.C.: Characterizing the remote control behavior of bots. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 89–108. Springer, Heidelberg (2007)
30. Wang, Y., Roussev, R., Verbowski, C., Johnson, A., Wu, M., Huang, Y., Kuo, S.: Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In: Proceedings of LISA (November 2004)
31. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of ACM CCS, Alexandria, VA (October 2007)