Cyber Grand Challenge (CGC) Monitor - A Vetting System for the DARPA Cyber Grand Challenge

By

Michael F. Thompson, Timothy Vidas

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2018 USA

Providence, RI (July 15th - 18th)

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

Cyber Grand Challenge (CGC) monitor: A vetting system for the DARPA cyber grand challenge

Michael F. Thompson [a, *], Timothy Vidas [b]

[a] Naval Postgraduate School, 1 University Circle, Monterey, CA 93943, USA
[b] Secureworks, One Concourse Parkway, Atlanta, GA 30328, USA

Keywords:
Full system simulation
Digital forensics
Proactive forensics
Introspection
Vulnerability analysis

A B S T R A C T

The DARPA Cyber Grand Challenge (CGC) pit autonomous machines against one another in a battle to discover, mitigate, and take advantage of software vulnerabilities. The competitors repeatedly formulated and submitted binary software for execution against opponents, and to mitigate attacks mounted by opponents. The US Government sought confidence that competitors legitimately won their rewards (a prize pool of up to $6.75 million USD), and competitors deserved evidence that all parties operated in accordance with the rules, which prohibited attempts to subvert the competition infrastructure. To support those goals, we developed an analysis system to vet competitor software submissions destined for execution on the competition infrastructure, the classic situation of running untrusted software.

In this work, we describe the design and implementation of this vetting system, as well as results gathered in deployment of the system as part of the CGC competition. The analysis system is implemented upon a high-fidelity full-system simulator requiring no modifications to the monitored operating system. We used this system to vet software submitted during the CGC Qualifying Event, and the CGC Final Event. The overwhelming majority of the vetting occurred in an automated fashion, with the system automatically monitoring the full x86-based system to detection corruption of operating system execution paths and data structures. However, the vetting system also facilitates investigation of any execution deemed suspicious by the automated process (or indeed any analysis required to answer queries related to the competition). An analyst may replay any software interaction using an IDA Pro plug-in, which utilizes the IDA debugger client to execute the session in reverse.

In post-mortem analysis, we found no evidence of attempted infrastructure subversion and further conclude that of the 20 vulnerable software services exploited in the CGC Final Event, half were exploited in ways unintended by the service authors. Six services were exploited due to vulnerabilities accidentally included by the authors, while an additional four were exploited via the author-intended vulnerability, but via an unanticipated path.

1. Introduction

Proactive forensics often blurs the lines between traditional forensics, pedantically requiring the application of a legal system, and similar techniques that may not ever be used to a legal end. In the digital space, such processes are increasingly common, with the use of similar, sometimes identical tools and procedures as those found in digital forensics, but to pursue a wide range of situations from corporate policy violations to complex computer administration troubleshooting. In many cases, adopting the relatively high standards demanded by legal systems early, even before demanded by circumstance, leads to not only a smooth transition into a digital forensics case, but in some cases the a priori action enables a case to form that otherwise might be impossible. For instance, if no digital artifacts were created as a result of some computer crimes, prosecution may be difficult, indeed in some cases the offense might even go unnoticed. However, observation and prosecution may be straightforward if the entity had previously put in place proper logging, netflow collection, and/or host-based software security agents. The level of preparation any entity might undergo varies drastically, with some proactively collecting data that may eventually become evidence (Shields et al., 2011), others physically installing hardware to enable future evidence collection (Carrier and Grand, 2004), or preparing tools and procedures in effort to achieve a state of readiness (Rowlingson, 2004).

* Corresponding author.
E-mail addresses: mfthomps@nps.edu (M.F. Thompson), tvidas@secureworks.com (T. Vidas).

https://doi.org/10.1016/j.diin.2018.04.016

The US Department of Defense Advanced Research Projects Agency (DARPA) created the Cyber Grand Challenge (CGC) to push the boundary of technology in "autonomous cyber defense capabilities that combine the speed and scale of automation with reasoning ability exceeding those of human experts (DARPA, 2016)." The challenge was framed around vulnerabilities in binary software. To encourage focused research, a concentrated, representative software environment was created: the DARPA Experimental Cyber Research Evaluation Environment, or simply DECREE. Concomitant with this focused environment, and due to the competitive nature of the challenge, many architectural and implementation design decisions made throughout program aimed to ensure the highest standards of integrity (Vidas et al., 2017). The mechanics of the challenge, in particular the final portion of the competition, was modeled heavily after attack-defend style cybersecurity Capture-the-Flag (CTF) competitions. The competition was structured as a "brokered" environment. That is, instead of competitors directly administering the hosts that required defense, or directly leveraging offensive actions upon one another, competitors uploaded software and minor metadata on how the software should be used (e.g. targeting information) to competition infrastructure services. This mediation afforded many desirable properties, among them more organizer control over the competition operations and the ability to catalog and inspect every input into the competition.

The CGC infrastructure development team built an analysis system to vet competitor submissions as one piece of a broad strategy to protect the competition integrity. The challenge organizers did not expect any of the CGC Final Event (CFE) competitors to attempt to subvert the competition infrastructure. Even so, there was desire for convincing evidence supporting the assertion that competition integrity was not compromised in violation of CGC rules (DARPA, 2016), and in the event that competition forensics were required, the unusual environment and relative urgency for results required investigative tools and processes to be foresight, not afterthought.

The competition consisted principally of two events, a qualifying event and for those that progressed, a final event. Participation in the CGC Qualifying Event (CQE) was open to any applicant who met a relatively open set of criteria (DARPA, 2014), and this accessibility motivated the vetting of CQE submissions. The analysis system, known as the *CGC Monitor* is built upon a full system simulator. An early goal was to simulate the entire competition infrastructure software execution environment and execute competitor-provided software within the simulated system prior to its introduction onto the actual competition infrastructure. The simulator is instrumented to detect attempts to compromise the operating system execution control paths or its data structures, (e.g., credentials used to identify a process and its permissions). For CQE, the goal of vetting all software prior to its reaching the infrastructure was realized. For CFE, time constraints required vetting of competitor software concurrently with execution on the actual competition infrastructure. Even though this vetting was not a prerequisite for introducing software into the competition, all submitted software was vetted prior to the naming of the CFE winners.

In addition to automated vetting of competitor-supplied software, an analysis system was developed to facilitate investigation of any executed competition submission. That is, an analyst can investigate any particular execution that occurred during the contest using a novel IDA Pro debugger client alongside an IDA-Python plug-in (Eagle, 2008). The analysis tool was available to further investigate sessions flagged during automated vetting, and was utilized after CFE to better understand which flaws were successfully exploited during the competition. The analyst tool includes reverse execution, automated identification of successful proofs of vulnerabilities (PoVs), and the ability to bookmark and return execution to points of interest.

During CQE, vetted submissions included each replacement challenge binary (RCB), and each proof of vulnerability (PoV), which in CQE was an XML file that controls network traffic sent to vulnerable Challenge Binaries (CBs). For CFE, every PoV was run against the RCB and IDS rule-set pair specified by the Cyber Reasoning System (CRS), that submitted the PoV. And every RCB was exercised using a sample of the service polls created for that CB. The competition infrastructure provides CBs with a repeatable source of simulated entropy, potentially leading to divergent execution between two sessions for the same service poll. The entropy sources used within the competition infrastructure were re-used in the simulated system to align code paths followed on each system.

For the purposes of this paper the term "malicious" is defined as attempts to subvert the competition infrastructure or scoring system, ultimately in violation of one or more of the CGC Rules (DARPA, 2016). The remainder of this paper first describes the simulated architectures in section 2 and the implementation of the CGC Monitor using the Simics full system simulation including strategies for detecting specific malicious activity in section 3. The analyst support functions and implementation of reverse execution features are then described in section 4. Details from at scale experiments of the CGC Monitor are provided in section 5. Experimental results are presented along with a discussion of performance in section 6. Finally, related work is described in section 7 and concluding remarks are provided in section 8.

2. Emulated architecture

CQE scoring occurred on a set of isolated virtual machines (VMs), each of which ran a single session, e.g., a reference PoV thrown against a competitor-provided RCB. Each VM included the two primary infrastructure software components: a player that replayed XML files to simulate client network traffic, and a launcher that spawned RCBs or CBs. CQE did not include an IDS component (DARPA, 2016). The CGC Monitor for CQE was similarly constructed, with individual simulated computers running the very same player and launcher software to replay CGC sessions.

Each team in the CFE had its own defended host upon which all of its services executed. When a team's CRS submits a RCB, that program eventually executes on that team's defended host. Each team also has its own PoV Thrower, which is a server that executes the PoVs submitted by that team's competitors, specifically targeting the team. Alongside the PoV thrower is a poller that sends service polls to the team's defended host and assesses responses to ensure the team's services are functional. All traffic that flows between a team's defended host and its PoV thrower and poller passes through an IDS. A CRS can submit IDS filters to block or modify traffic flowing to the services executing on a defended host. Each team's suite of components included a negotiator with which an executing PoV negotiates attributes of the proof of vulnerability. These negotiated attributes include whether the PoV is Type 1 (controlled crash), or Type 2 (memory disclosure).

For CFE, the system simulated by the CGC Monitor includes three distinct simulated computers that correspond to specific per-team servers in the competition infrastructure: the defended host; the IDS; and, the PoV thrower. To reduce the quantity of simulated computers, service polls were originated on the simulated IDS server rather than a separately simulated server. Similarly, within the simulated system, the negotiation service runs on the IDS rather than a distinct server. The simulated computers run the exact operating system, (and custom hypervisor), deployed on the competition infrastructure and the exact services, e.g., the programs that launch CBs and PoVs. While the CFE competition infrastructure computers include multi-core processors, the simulated computers are configured as single-core processors because these are more efficient for Simics to simulate, and none of the deliberately planted vulnerabilities in CGC relied on multiprocess race conditions. (The CGC Monitor is configurable to use multi-core processors and to run many CGC sessions concurrently on a single set of simulated computers, and this was done to help identify a race condition in the DECREE infrastructure kernel during its development.)

As in the CFE infrastructure, the simulated systems include ethernet links dedicated for in-competition data and separate links for exchange of control information.

3. CGC monitor design and implementation

The design goal for the CGC Monitor was to monitor all CGC infrastructure software that could be compromised by malicious competitor submissions. This naturally includes the operating system that hosts executables provided by a CRS.

The range of strategies considered for monitoring the CGC included, at one extreme, conducting the entire competition upon a simulated system with real-time introspection into the infrastructure software. Another option was to conduct the competition on real hardware, and then repeating the entire competition on a simulated system. This second run of the competition would have been performed using a record/replay tool such as Panda (Dolan-Gavitt et al., 2015), facilitating inspection and analysis of competitor software through iterative replays, looking for signs of malicious activity. After trivial analysis it was apparent that true real-time monitoring of CGC infrastructure through introspection would lead to competitions with very long durations, for reasons summarized in section 6.1. Nevertheless, the competition infrastructure team held to the general goal of approaching real-time monitoring, and sought a design that could play individual CGC sessions and monitor events as they happened.

The Simics full system simulation is the core of the monitoring system. Simics includes a simulated Intel-based computer platform that broadly corresponds to those deployed in the CGC infrastructure. All monitoring is implemented by functions that non-intrusively observe the state of the simulated system. No instrumentation exists within the simulated execution environment, which is monitored using scripts developed for use with the Simics Hindsight product (Wind River, 2015c). The only portions of the CGC Monitor that execute within the simulated environment are functions to support coordination, such as the transfer of CGC competitor submissions into the execution environment, and invocation of the CGC competition infrastructure services, (e.g., the launcher that spawns CBs).

Though Simics also includes an "Operating System Awareness" feature to aid in monitoring selected Linux kernel environments, use of that feature significantly degrades performance by disabling hardware assisted acceleration. Because of that, and because Simics OS awareness does not include FreeBSD, the CGC Monitor required development of a novel OS awareness function using the basic Simics building blocks of breakpoints and callbacks.

3.1. CGC monitor OS awareness implementation

As a brief introduction to the strategy, consider a full system simulation environment, (e.g., Simics) that allows a developer to set breakpoints on selected memory accesses, and associate developer-programmed callbacks with each breakpoint. Setting a "modify" breakpoint on the memory address at which the kernel maintained a pointer to the currently scheduled thread causes the associated callback to be invoked whenever the scheduled thread changed. Depending on the context referenced by the current thread pointer, the callback sets additional breakpoints to monitor actions taken while that specific thread is scheduled, e.g., setting an "execute" breakpoint on addresses that handle syscall entry into the kernel to invoke an associated callback each time a process makes a system call. Similarly, catching invocations of `execve`, (by breaking on execution of the address of that function in the kernel), discovers parameters passed when selected processes are started, facilitating the tracking of the execution of specific programs. This strategy is sufficient because `execve` is the only mechanism within the DECREE execution environment that causes execution of programs of interest. These breakpoint/callback primitives are tied to addresses taken from kernel link maps, and offsets calculated from kernel internal data type declarations.

The CGC Monitor uses three types of Simics callbacks other than the breakpoint callbacks described above. These callback types are:

- *Processor exceptions* are used to monitor events raised by the CPU. Monitoring page fault exceptions provides the analysis function with the execution address at which a segmentation fault occurs. An execution breakpoint within the kernel's signal handler generates a callback when the kernel generates a signal. If the signal type reflects a segmentation fault, then the previous page fault for that process is assumed to have caused the signal. Page faults are also monitored by analysis functions to identify when physical memory is about to be mapped to a process's address space, thereby enabling the setting of breakpoints on physical memory in user mode as soon as the kernel returns to user mode i.e., after the kernel has mapped the referenced memory. Illegal instruction (`SIGILL`) faults are recorded directly by these callbacks, without reference to operating system instrumentation, other than to identify the offending process.
- *Mode change* callbacks are used extensively in the Simics Hindsight product's OS awareness feature, contributing to the aforementioned performance degradation. Use of these callbacks causes Simics to disable its "VMP" kernel module, which uses hardware virtualization to run much of the simulation natively on the host processor. The CGC Monitor uses these callbacks sparingly, e.g., during the window of `execve` processing to catch the first return to user mode within a process. Mode change callbacks are also used as described above to catch returns to user mode after physical memory is mapped following a page fault.
- *Simulation stopped* callbacks are invoked when the simulated execution stops, e.g., because a breakpoint is reached for which there is no corresponding callback. These callbacks are utilized within analysis functions in support of reverse execution. Simics breakpoint and mode change callbacks are not generally useful when executing in reverse (as described in 4.3). If breakpoints are set without corresponding callbacks, reverse execution will find the most recent breakpoint, stop execution at that point, and execute the corresponding "simulation stopped" callback. This is done in lieu of breakpoint callbacks.

3.1.1. Breakpoints on linear vs physical addresses

Simics x86 processor models support the setting of breakpoints on physical addresses and "linear", i.e., virtual, addresses. The monitor uses linear address breakpoints when monitoring the kernel. When monitoring user space for analysis functions, CGC Monitor employs physical address breakpoints because linear addresses are shared between processes. Use of physical address breakpoints requires the monitor to detect the mapping of physical pages, which it does by catching processor page fault exceptions as described earlier. DECREE provides no interfaces via which a process could cause remapping of page table entries or copy-on-write, thereby simplifying the management of breakpoints on physical memory. Support for monitoring a more general execution environment would require suitable extensions to the breakpoint management.

3.2. Fidelity

Vetting software with a simulator requires code paths followed on the simulated system to match those on the target system. If malicious software can detect differences between the target system and the simulated system, the software can mask its malicious intent when running on the simulated system. The limited number of system calls within DECREE reduces the opportunities for programs to identify differences in the two environments. Additionally, the use of a custom hypervisor allows DECREE to produce identical responses to machine instructions that might otherwise reflect hardware differences (Vidas et al., 2017). At least one competitor demonstrated the ability to detect the difference between DECREE executing on CFE competition infrastructure and DECREE within the publicly available VM, and they used this detection capability to diverge execution of their RCBs between the two platforms. The CGC Monitor execution paths purposefully only matched those of the CFE competition infrastructure.

Simics employs processor models to emulate execution of code on specific processors. The fidelity of the processor model to the physical processor is critical to avoiding potential execution divergence. DARPA previously contracted with Intel to develop a suite of application monitoring tools, and while those tools did not become part of the CGC Monitor, the CGC Monitor did incorporate the high-fidelity processor model from that project (Intel Corporation, 2015). This processor model replaced the processor model within the standard Simics "Intel Core i7" model library.

Simics processor models simulate time such that all instructions take a single unit of time, i.e., one clock cycle, to run. DECREE processes have no direct ability to observe the passing of clock cycles, however this might result in kernel scheduling divergence (e.g., differences in how many instructions get executed during a specific time slice). Timing differences between the simulated system and the target environment might be exploited to distinguish between the two environments. While DECREE does not give a process access to an explicit clock, processes do have opportunities to observe some implications of time passing. For example, the `receive` call may return different quantities of bytes on different implementations. Another opportunity for divergence occurs because some CBs contain multiple binaries that execute concurrently. Results of IPC operations (transmit/receive using shared file descriptors) can vary due to differences in kernel scheduling. For example, one process may transmit a large number of bytes using a shared descriptor. The number of bytes seen during a single receive operation performed by another process may depend on how long the transmitting process was scheduled.

3.3. CGC monitor operating systems

As noted earlier, the choice to not rely on the limited Simics OS Awareness support was driven by more than just performance considerations. While CQE was hosted on a 32-bit Linux kernel (similar to those supported by the Simics OS Awareness), the DECREE execution environment includes a set of system calls that differ from those provided by Linux. Additionally, CFE was hosted on a 64-bit FreeBSD kernel, (that includes the 32-bit DECREE execution environment.) The CGC Monitor is configurable to simulate systems containing 32-bit or 64-bit versions of either operating system, including a heterogeneous mix within the same monitored system.

3.4. What is monitored?

Kernel execution is monitored for signs of malicious behavior while any of the following processes are scheduled: RCBs; PoVs; and, IDS software (because it consumes competitor-provided IDS filter rules). Signs of malicious behavior include: instances of return oriented programming (ROP), process credential manipulation, writing to kernel page table structures, and unexpected execution of code within the kernel, (e.g., `execve` functions while a challenge binary is scheduled), or execution at any address outside of defined text sections. The set of monitored actions resulted from discussions with veteran CTF players and individuals with experience subverting platforms.

- *Kernel ROP* Execution of a `ret` instruction in kernel mode whose destination address does not follow a `call` instruction. This is implemented by setting execution breakpoints over the entire kernel executable linear address space, qualified via the Simics set-prefix function, which masks the breakpoints to ignore any instructions whose mnemonic does not begin with a given string. Thus, execution of each `ret` instruction in the kernel generates a callback to a function that inspects the destination address to ensure it follows a call. DECREE infrastructure kernels are configured to use address space layout randomization (ASLR). The breakpoint range is associated with addresses of kernel module code by recording the random kernel address ranges during a provisioning step which creates a snapshot of a booted system image that then serves as the baseline for all instances of the monitor. Kernel ROP monitoring is only engaged while targeted processes are scheduled.
- *Kernel Page Tables* Page directories and table modification is detected via modify breakpoints set on the linear address of pages containing these structures. Transitions to use different operating systems with different page management schemes, e.g., Intel physical address extensions (PAE), resulted in several variations of this monitoring.
- *Credentials* Modify breakpoints are set on the pointers to user credentials within the thread structures of monitored processes, and on the structures containing the credentials.
- *Unexpected Execution* Sections of kernel code that were not expected to execute during PoV or RCB sessions were monitored with execute breakpoints. These kernel text sections were identified manually based on coarse analysis of functions an attacker would value.

4. Analyst support functions

In addition to automated vetting of competitor submissions, the CGC Monitor includes functions to support investigation of competitor submissions as they execute during simulated competition sessions. These functions use the very same simulated architecture as the vetting functions.

A goal of developing the analysis functions was to aid in determining the root causes of foul play or successful PoVs. The existence of a successful PoV in CFE simply reflects that there is a vulnerability, but does not indicate which vulnerability, (e.g., some PoVs had more than one), or the path followed to exploit the vulnerability. The challenge is related to the real world situation where a fuzzer generates a program input that causes a crash, leaving an analyst to find the bug that created the vulnerability.

4.1. Capturing session events

When enabled, analysis functions record all system calls made by a binary executing on DECREE. The CGC Monitor records all parameters, including data read and written with receive and transmit calls. The system can optionally generate a full instruction trace, including data accesses, for any session. CGC-specific events are automatically caught and recorded as they occur, including:

- Generation of operating system signals, e.g., a segmentation fault that occurs as part of a Type 1 PoV.
- Negotiation of PoVs. PoV interaction with the negotiator, including negotiated register values and submission of Type 2 protected memory values.
- Type 2 PoVs. The record of these events includes execution addresses at which the memory was read.
- Execution of `ret` instructions whose target address does not follow a call, reflecting potential use of ROP by a PoV.
- Execution of any address outside of defined text sections or memory allocated as executable.

4.2. Interactive program analysis with reverse execution

The analyst interacts with a simulated CGC competition session under control of the IDA Pro GDB debug client. This “IDA Client” was extended using IDAPython scripts to interact with the CGC Monitor, supporting reverse execution to breakpoints and events of interest. A typical IDA Client session begins with the analyst naming a session to replay. The CGC Monitor executes the session and automatically pauses the simulation at a point of interest, e.g., reading protected memory as part of a Type 2 PoV. The IDA Pro interactive disassembler launches and connects to the CGC Monitor's GDB server functions, displaying the execution point at which the simulation paused. The analyst can then use standard IDA debugger commands to interact with the program. The tool supports extended hot-key and menu selections to include operations such as shown in Table 1.

The analyst can set and jump to bookmarks anywhere within the session. Initial, pre-generated bookmarks include the point at which the process begins execution, the point at which the simulation was initially paused, and events such as the first execution of an address within the stack. The final item in the above list of operations generates a set of bookmarks, one for each instruction that transfers the tracked value from/to memory and registers. The operation is similar to taint tracking, only in reverse and it halts upon any manipulation that is not a strict load, move or indirect memory transfer. An example use case tracks the providence of a faulting return address on the stack. When analyzing successful CFE PoVs, this operation will usually halt at the receive system call that reads the address from the PoV, leaving a series of bookmarks denoting execution points at which the faulting address was copied. This sequence of bookmarks, starting at the fault and ending at the ingest of the faulting address, gives the analyst specific locations in which to look for memory corruption.

The analyst support function includes an automated operating mode that runs without an IDA client, and creates reports of events associated with a successful PoV session. These reports summarize events in the session, e.g., execution from the stack region, and include the reverse data trace bookmarks. This capability was used to create reports for all of the successful CFE PoVs, and the results can be seen in the archive of the CGC CFE corpus (Caswell, 2017).

Table 1
Analyst tool hotkeys. Convenience key combinations for investigation using the IDA Pro tool described in section Interactive program analysis with reverse execution.

Hotkey	Operation
Alt-Shift-F9	reverse
Alt-Shift-F8	reverse step over
Alt-Shift-F7	reverse step into
Alt-Shift-F4	reverse to cursor
Alt-F6	reverse until just before current function is called
Alt-Shift-r	reverse to previous write (selected register)
Alt-Shift-a	reverse to previous write (selected address)
Alt-Shift-s	reverse to previous write (selected stack address)
Alt-Shift-o	jump to initial debug eip (just before fault)
Alt-Shift-t	jump to start of process
Alt-Shift-p	set an execution bookmark
Alt-Shift-j	jump to a bookmark (chosen from list)
Alt-Shift-u	run forward until in user space

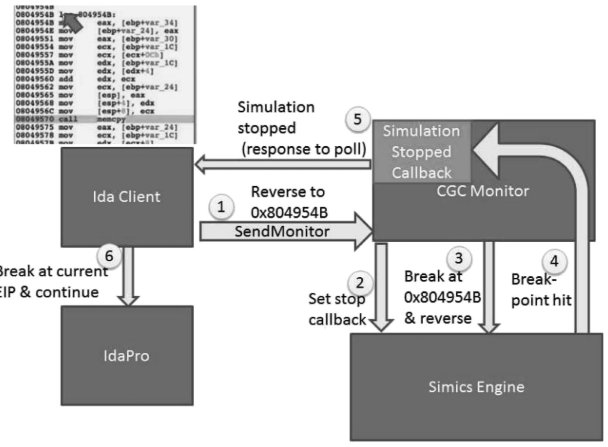
4.3. Debugger implementation considerations

Simics includes a GDB server function via which remote GDB clients can control the simulated execution of processes. When used for analysis of applications, the GDB server is typically combined with Simics OS awareness functions to maintain the context of the target process. As previously noted, the CGC Monitor does not use Simics OS awareness functions, and thus it maintains the necessary process context information to ensure that the state of the proper process is returned to the GDB client.

Simics reverse execution support significantly slows down the simulation and is only enabled for IDA Client sessions. The CGC Monitor enables the feature once the target process begins execution. Simics creates the appearance of reverse execution by recording a series of “micro checkpoints” that are referenced during any reverse operation to move the state of the simulation backwards in time. The actual backwards progression of the simulation is not strictly serial, and there is no specification of the sequence or frequency at which callback events might be encountered. For example, if a breakpoint is set at an executable address five instructions prior to the point at which the reverse operation commences, any callback associated with that breakpoint may be invoked several times before Simics identifies the correct, (i.e., most recent), encounter with the breakpoint address. While the Simics reverse operation does not respect the ordering of callbacks, it does eventually identify the proper simulated execution state for breakpoints. If the breakpoint in the proceeding example had no associated callback, then the simulation will stop after reversing to the most recent state at which the breakpoint memory address is the EIP for any process. Thus, prior to invoking the Simics reverse operation, all callbacks are removed, and a “Simulation Stopped” callback is added.

The IDA Pro GDB client (as of version 6.8) does not include support for reverse execution. However, IDA Pro can be extended via scripts and plug-ins. Support for reverse execution within the IDA debugger client was added using IDAPython. Reverse operations are out of band with standard client/server GDB communication. The IDA GDB client includes a `SendGDBMonitor` function that forwards a given command to the GDB server. The Simics GDB server simply passes these to the Simics command interpreter. Since the CGC Monitor is treated by Simics as a nameable object, its methods can be invoked by the Simics command line arguments, thus CGC Monitor functions can be invoked remotely by IDAPython scripts.

To illustrate the flow, imagine the analyst wishes to reverse execution to an address indicated by the current cursor position. The sequence of events are shown in Fig. 1. First, the IDA Client plug-in uses `SendGDBMonitor` to send a command that causes the CGC Monitor to be invoked via a function that reverses execution to the given address. The function is implemented by defining a stop callback, then setting a breakpoint at the given address and directing Simics to reverse. When the breakpoint is reached, Simics stops the simulation. When the simulation stops, the stopped callback executes and checks if the simulation has reached the



```
0804954B l   804954B:
0804954B m          eax, [ebp+var_34]
0804954E mov        [ebp+var_24], eax
08049551 mov        eax, [ebp+var_30]
08049554 mov        ecx, [ebp+var_1C]
08049557 mov        ecx, [ecx+0Ch]
0804955A mov        edx, [ebp+var_1C]
0804955D mov        edx, [edx+4]
08049560 add        edx, ecx
08049562 mov        ecx, [ebp+var_24]
08049565 mov        [esp], eax
08049568 mov        [esp+4], edx
0804956C mov        [esp+8], ecx
08049570 call       memcpy
08049575 mov        eax, [ebp+var_24]
08049578 mov        ecx, [ebp+var_1C]
```

Simulation stopped (response to poll)
5
Simulation Stopped Callback
CGC Monitor
Ida Client
1
Reverse to 0x804954B
SendMonitor
6
Break at current EIP & continue
2
Set stop callback
3
Break at 0x804954B & reverse
4
Break-point hit
IdaPro
Simics Engine

Fig. 1. *The flow sequence to reverse to cursor address.* The IDA Client directs Simics to reverse execution (1). In turn, the CGC Monitor defines a stop callback (2) and sets an address breakpoint (3) at the address specified in step 1. Simics processes execution until the breakpoint is reached at which point simulation is stopped (4), triggering the stop breakpoint set in step 2, which verifies that simulation has reached the desired address. The IDA Client, which after step 1 polls the CGC Monitor, receives a status update that Simics has reversed execution to the desired addressed. The IDA Client then sets a breakpoint at the current EIP and starts execution (6) which forces (via a GDB IDA Pro function) all the register states to be updated in IDA Pro.

desired state, and if so, it updates a status message that will be read by the client. The `SendGDBMonitor` function is not synchronous, so while the CGC Monitor is coordinating the reverse execution, the IDA Client polls the CGC Monitor, again with the `SendGDBMonitor` command, but this time via a function that returns the status message maintained by the CGC Monitor. Once the polled message indicates that the simulation has stopped, the IDA Client plug-in calls the standard GDB IDA function to set a breakpoint on the current EIP, and run forward. This final step forces the IDA GDB client to query the GDB server for an update on the current register state, thus presenting the correct execution state to the analyst. To our knowledge, no other method exists to force the IDA GDB client to update its representation of the target register state, and so this trick of setting a breakpoint at the current EIP and continuing is employed throughout the tool. If a provided address is not encountered (meaning a faulty address was provided in step 1 - one that had not been previously executed), the CGC Monitor halts reverse execution at the very first recorded user space instruction and breaks in an error condition.

5. Scaling and evaluation

The CGC Monitor was designed to vet submissions in a CFE simulation consisting of 100 roughly five-minute rounds having at most twenty services (CBs) per round, where seven teams submit replacements for each service and PoVs targeting each of the other six teams. This leads to a potential total of 14,000 different RCBs, with 140 concurrent per round, and up to 120 different PoVs per round, for a total of 12,000 PoVs. To meet this load, the CGC Monitor can scale to hundreds of concurrent emulated instances of CGC systems. Each instance consists of a Simics simulation of one or more emulated computers (e.g., each simulation of CFE includes three networked computers per instance), and each simulates a single CGC session at any time.

Two blade enclosures host the CGC Monitor, each enclosure contains 16 blade servers having a single quad core Intel Xeon processor running at 3.5 GHz and 4 GB of RAM. Twenty-nine blades each hosted six instances of emulated systems, and three hosted three instances each, (these three blades were shared with the application coordination service). The hardware performance determines the duration required to emulate the set of sessions, it does not affect the execution of CBs.

During CFE, CPU utilization varied depending on properties of the sessions being simulated. Some of the more pathological sessions described in section 6.1 led to sustained CPU utilization of 100% on the individual blade core allocated to the session. Even though a blade core load would reach 100%, the queuing system described below ensured that all sessions were eventually evaluated. Memory usage did not exceed 50%, and did not vary noticeably between sessions.

Distributed synchronization, configuration management, and low level queuing functions are implemented with the Apache ZooKeeper application coordination service (Hunt et al., 2010). ZooKeeper is installed as a three node cluster on three of the blade servers, and CGC Monitor relies on this clustering to provide redundancy and fault tolerance for the automated vetting results. Authoritative results of each CGC session are written to individual ZooKeeper nodes, and are eventually transferred to a MySQL database. A distinct “Monitor Slave” manages each Simics simulation instance, using a ZooKeeper client to lock and retrieve CGC sessions for simulation within its associated Simics session. Each blade server contains one or more Monitor Slaves, one per Simics session. All monitor slaves draw CGC sessions from a common pool, maintained in a ZooKeeper node hierarchy. A single “Monitor Master” populates this pool with CGC session data obtained from the CGC competition infrastructure. The Monitor Master copies competitor submissions, e.g., RCBs, into a forensics repository mounted via NFS by the other blades. Metadata about the sessions (e.g., entropy seeds) are stored in ZooKeeper nodes created by the Master for each session.

Blade servers can be dynamically added and removed while the CGC Monitor runs. Newly added blade servers automatically reference ZooKeeper nodes to retrieve their configuration and begin processing GCG sessions. Any session initiated but not completed by an absent Monitor Slave is returned to the pool for processing by some other Monitor Slave.

Each Monitor Slave also uses ZooKeeper nodes to coordinate with CGC Monitor functions executing within its associated simulated environment to spawn the CGC competition infrastructure services, e.g., the CB launcher. These ZooKeeper clients obtain their configuration information and session replay metadata from ZooKeeper nodes. Programs within the simulated environment communicate with real world computers, (e.g., to ingest competitor submissions or to create a ZooKeeper node to signal completion of a session), using the Simics real network interfaces (Wind River, 2015b). Each emulated computer includes multiple emulated Ethernet interfaces, one of which is dedicated for this communication with real world processes.

Simics optimizes simulations by detecting and skipping idle loop processing. As a result, running ZooKeeper clients within the simulated environment presents interesting implementation challenges because the speed of clocks in those environments vary drastically from the real-world clocks referenced by the ZooKeeper server cluster. During a CGC replay having little processing, or while idle, the clock in the simulated system may run several times faster than the clocks in the server cluster and, during a CPU-intensive replay, (e.g., a CB looping on a failed system call), the simulated clock may run hundreds of times slower than the clocks in the cluster. One side or the other of the ZooKeeper client-server protocols will often see session expirations or timeouts. Fortunately, ZooKeeper employs robust session management reporting within the client, enabling the client to reliably detect the loss of, and subsequently reinitiate, a session. Some CGC replays experience several ZooKeeper session timeouts in the course of the replay, but are able to recover with no loss of data or noticeable performance degradation resulting from frequent session reestablishment.

6. Results

The primary objective of the CGC Monitor was to identify and subsequently analyze any attempt to undermine the competition infrastructure. No team submissions were found to have been designed to subvert the CGC competition infrastructure.

During CFE, the CGC Monitor did flag 41 occurrences of "returns without calls" (ROP) within the kernel. Each instance occurred at the same location in the freeBSD kernel, specifically within a return to the fork trampoline function (`cpu_switch.S`) during creation of a RCB process. This occurred with several different multi-binary challenge sets, in about three percent of the sessions for those challenge sets. CGC referees determined these events were false positives. That is, the intended execution of this kernel trampoline function is to return to a code sequence that had not made a call. The CGC Monitor should not have monitored execution of these events because they occurred during fork processing rather than during execution of an RCB.

Following CFE, and not pertinent to the integrity of the competition, the CGC Monitor was used to analyze competitor submissions. Each CFE vulnerable software service had one or more reference PoVs, patched versions of the services, and descriptions of how their vulnerabilities could be exploited all provided by the service author. In this post-CFE analysis, the objective was to identify which vulnerability was exploited. Though some services include multiple vulnerabilities, no more than one vulnerability in any given service was exploited during CFE. Of the 20 services exploited in CFE, (listed in Table 2), 10 were exploited in unintended ways. Six (30%) were due to vulnerabilities that were not intended by the service author. Further, four (20%) of the exploited services were exploited through paths not identified by the author, and these paths were often much simpler than those envisioned by the author. For example, the `CROMU_00055`[1] service had a deliberate heap management bug that returns a buffer of size zero in response to a request for a buffer of size `0xff+1`. The author's documentation describes an exploit that carefully constructs data structures within the heap, which are then used to corrupt the execution control path. The competitor systems did not deploy heap exploits for this service, rather, they discovered the zero length buffer led to direct disclosure of values copied from protected memory.

Table 2
Author Intention of Exploited CFE Services: Of the 20 exploited services in CFE, half were exploited in ways unintended by the author of the vulnerable service. Six services included a vulnerability unknown to the author, yet discovered by competitors, and an additional four vulnerabilities found were intended, but exploited via unintended means.

CBID	As intended	Description
CROMU_00046	Y	
CROMU_00051	N	unintended path
CROMU_00055	N	unintended path
CROMU_00058	Y	
CROMU_00064	Y	
CROMU_00065	N	unintended path
CROMU_00073	N	unintended path
CROMU_00088	N	unintended vuln
CROMU_00094	Y	
CROMU_00095	N	unintended vuln
CROMU_00096	N	unintended vuln
CROMU_00097	N	unintended vuln
CROMU_00098	Y	
KPRCA_00065	N	unintended vuln
KPRCA_00094	Y	
NRFIN_00052	N	unintended vuln
NRFIN_00059	Y	
NRFIN_00063	Y	
YAN01_00015	Y	
YAN01_00016	Y	

6.1. Performance

The CGC Monitor is configurable to monitor different types of events, and in general, more monitoring results in slower simulation of CGC sessions. Vetting does not require monitoring of application or user space operations, e.g., syscalls need not be logged to vet competitor submissions. However, syscall logs are convenient to have when reviewing competition activity. Similarly, monitoring of PoV negotiation and access to protected memory is not necessary for vetting. But, again, it is convenient to have those results without having to separately rerun sessions. Since it was previously established that vetting would be concurrent with CFE, and not a prerequisite to running submissions on CFE competition infrastructure, additional application-level monitoring was enabled at the cost of longer simulated sessions. The additional logging allowed referees to compare PoV success as seen on the analysis system with that seen in the actual competition, and confirm the consistency of those results.

Vetting during CFE was fully automated, though humans monitored the results and were able to adjust the monitoring based on system performance. Using the monitoring configuration selected for CFE, many sessions completed vetting within a few seconds. Some would take days to complete, (if monitoring continued for the duration of the simulated session). Some CGC sessions are long-running due to several conditions, some of which led us to prematurely terminate monitoring of some sessions:

1. The Simics product includes an optional *Accelerator*, intended to improve the performance of multi-computer simulations (Wind River, 2015a). The Accelerator uses multithreading to run multi-machine simulations in parallel on multiprocessor hosts. Experiments with simulations of CGC sessions showed no performance improvement when using the Accelerator. Simics includes several tunable parameters to optimize performance of multi-threaded simulations, including the latency controlling how frequently threads are synchronized. No tested configuration offered significant performance over the single threaded implementation which may well simply be a property of the CGC sessions being simulated.
2. *Errored Syscalls* A process spinning on calls to `receive` passing a closed file descriptor invokes the kernel many times in fifteen seconds (all CGC sessions are terminated after fifteen seconds). Large numbers of syscalls slow down the simulation. The CGC Monitor was configured to terminate monitoring after several hundred failed syscalls. This situation generally occurred when a competitor CRS incorrectly reformulated a RCB, resulting in the poller exiting and insufficient error handling within the CB communication protocol.
3. *Long running PoVs* PoVs may be more reliable if they execute for the duration of the session, because premature exiting can result in the target RCB failing to read some data that had been written by the PoV. Keeping a PoV alive by using `fdwait` to sleep for a long period would be the CGC Monitor's preference to keep a PoV executing because Simics skips noop cycles. Unfortunately, competitors had PoVs spin on syscalls or just spin in user space, which requires simulation.
4. *Denial of Service* Scores are affected by the wall-time required to complete polls. Thus, if a competitor PoV can consume resources

[1] The source and readme for this service can be found at https://github.com/CyberGrandChallenge/samples/tree/master/examples/CROMU_00055.

on another's defended host, that might increase the wall-time required to service polls on that defended host. One way to consume defended host resources is to cause an RCB to continuously read bytes, one at at time, from a PoV until the fifteen second limit is reached. In the most extreme example of this, a competitor threw a PoV that caused the RCB to read 1,188,880 individual bytes in 1,188,883 system calls. Those fifteen seconds of real-time require 62 h of monitored simulation time. (Turning off kernel ROP monitoring reduces the replay time for that session to eighty-four minutes.) That particular PoV did not score (other than perhaps indirectly by consuming resources on the defended host). A strategy followed by at least one team was to score with a PoV (direct benefit), and then enter an infinite loop (indirect benefit by hindering an opponent).

Attempts to slow down defended hosts by consuming their resources were so prevalent that about halfway into the competition it became clear that some sessions were consuming all CGC Monitor resources, with no visible progress being made. The CGC referees therefore reconfigured the system to only monitor events necessary for vetting, and they bounded execution. Roughly halfway through CFE, the starting configuration was altered to:

1. Terminate sessions after execution of 136,000,000 instructions (roughly fifteen minutes, session dependent).
2. Monitor only kernel events as listed in section 3.4.
3. Disable syscall logging

Immediately following CFE, changes 2 and 3 were rescinded (the instruction upper bound remained), and the remaining sessions were replayed through the CGC Monitor in order to secure independent confirmation of PoV success.

Competitive advantages gained by creating resource intensive executables, or subverting RCBs to consume resources, reduces the prospects for running a CGC-type capture-the-flag competition on simulated hardware. Even when all monitoring is disabled, some of the more pathological sessions required fifteen minutes to reach a fifteen second timeout. These cases appear to have resulted from large quantities of single byte transmits and receives.

6.2. Limitations

The CGC Monitor primarily screened for attempts to subvert the DECREE operating system. CGC Monitor did not monitor the IDS service in CFE in an IDS-specific way, (an application written in Go), though that service did consume competitor submissions in the form of IDS filter rule sets. The CGC Monitor did monitor the DECREE infrastructure kernel on the IDS host while the IDS service processed rule sets. Similarly, the CGC Monitor was not used to monitor the poller nor the DECREE kernel on the poller host, though the host consumed competitor data in the form of poll responses.

As noted in section 3.2, malicious functions could use timing differences to detect the use of Simics as a simulated vetting platform, and branch away from exhibiting symptoms of maliciousness. The use of Simics in CGC was a relatively closely held secret, which might not be the case if there were subsequent competitions. The authors see no solution to the problem of malicious software detecting timing differences between the simulated platform and the target hardware.

The CGC Monitor was written to act upon CGC binaries exclusively and therefore does not readily work with commodity software. However, Simics is a full system x86 simulator and the principles (and open source software) of the CGC monitor could be extended to work on software other than CGC binaries.

7. Related work

Significant research has been performed toward efficient and correct execution virtualization (Barham et al., 2003), simulation, replay and reversal (Engblom, 2012). Analogous to CGC monitor, systems like Aftersight (Chow et al., 2008) log nondeterministic inputs, in this case to Virtual Machines, in order to replay, forensically log, or parallelize execution at the machine level. Since Aftersight is implemented at the Virtual Machine Monitor, the recording and analysis can be performed regardless of the software stack — including the operating system. CGC Monitor benefits from similar operating system portability and can be expanded to work with systems beyond those employed in CGC.

Numerous dynamic analysis systems exist, often used commercially for “malware detonation,” or “malware forensics” (Deng et al., 2012) — as a “sandbox.” Such systems (Willems et al., 2007) typically report akin to forensics or incident response tools in that they instrument a commodity operating system and report upon the effects of executing either a suspicious or known-bad file. Anubis is an example of an academic dynamic malware analysis system built using the QEMU full system emulator. (Bellard, 2005; Bayer et al., 2009). Monitoring is performed outside of the analysis environment. The emulator performs execution monitoring by observing the execution of pre-computed memory addresses corresponding to system API functions. As with the Panda system mentioned in section 3, the reliance on QEMU leads to the system inheriting limitations of the emulator fidelity, and its vulnerability to anti-analysis techniques such as those deployed by competitors during CFE (Shellphish, 2017).

DRAKVUF is a dynamic malware analysis system designed to use Intel hardware virtualization extensions and the Xen hypervisor (Lengyel et al., 2014). DRAKVUF does not require agents to run on the monitored operating system, i.e., it monitors from outside the VM. It enables kernel monitoring using breakpoints and callbacks somewhat similar to those used within the CGC monitor, except that DRAKVUF injects breakpoints by writing `INT3` instructions directly into the virtual memory of the guest. The Simics support for setting breakpoints on simulated memory avoids any need for modifying guest memory. Ether is another dynamic malware analysis system that uses Intel virtualization hardware and the Xen hypervisor to facilitate execution tracing from outside of a VM, through use of `VMEntry` and `VMExit` to transition between the guest and the VMX root mode (Dinaburg et al., 2008). These hypervisor-based analysis systems are typically used to create traces that are then separately analyzed to detect malicious activity. In contrast, the CGC Monitor detects malicious activity during execution of the simulated session, which includes all networked components that contribute to the sessions.

8. Conclusion

The architecture and time constraints of the Cyber Grand Challenge prompted a unique investigative challenge grounded in established problem spaces. The competition itself sought to create a representative, but minimal environment for software attack and defense. Consequently, a need existed for proactive collection of competition records, and concurrent development investigative tools to maintain a state of readiness.

Monitoring CGC sessions on an instrumented full-system simulator ultimately provided confidence that software generated by competitors did not attempt to subvert the competition infrastructure. Choices made in the design of the competition (e.g., the use of wall time to assess the impact of the IDS) led teams to deliberately consume resources on their competitor's defended hosts, and this limited the ability for the instance of CGC Monitor to fully vet the competition in real time.

The Simics full-system simulator allowed the CGC Monitor to simulate the key CGC infrastructure components, including the exact CGC software stack. Limiting monitor design to the use of breakpoints and callbacks required that the CGC Monitor fully bridge the “semantic gap” without assistance from the simulator, but this also afforded the flexibility necessary to monitor a range of different operating systems. Further, the Simics support for reverse execution enabled the creation of an analyst support tool, facilitating investigation of suspicious sessions. In addition to automatically vetting most competitor software, the CGC Monitor analyst tool aided analysts in determining which particular vulnerabilities were exploited during the competition. For instance, in the CGC Final Event half of the vulnerabilities exploited were either gratuitous or accessed via unintended execution paths.

The CGC Monitor is available at https://github.com/mfthomps/cgc-monitor. Analysis results from CFE, generated by the monitor, are at https://github.com/mfthomps/CGC-Analysis.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency under Air Force award number FA8750-12-D-0005. The content is solely the responsibility of the authors and does not necessarily represent the official views of the United States Government.

The authors would also like to thank the many people that made the Cyber Grand Challenge possible, especially the Competition Integrity Team, and the anonymous reviewers for their time and feedback.

References

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A., 2003. Xen and the art of virtualization. In: ACM SIGOPS Operating Systems Review, pp. 164–177. ACM volume 37.

Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E., 2009. Scalable, behavior-based malware clustering. In: NDSS, vol. 9, pp. 8–11. http://anubis.iseclab.org.

Bellard, F., 2005. QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track.

Carrier, B.D., Grand, J., 2004. A hardware-based memory acquisition procedure for digital investigations. Digit. Invest. 1, 50–60.

Caswell, B., 2017. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/. http://www.lungetech.com/cgc-corpus/.

Chow, J., Garfinkel, T., Chen, P.M., 2008. Decoupling dynamic program analysis from execution in virtual environments. In: USENIX 2008 Annual Technical Conference ATC'08. USENIX Association, Berkeley, CA, USA, pp. 1–14. http://dl.acm.org/citation.cfm?id=1404014.1404015.

DARPA, 2014. Cgc Extended Application. URL: http://archive.darpa.mil/cybergrandchallenge_competitorsite/Files/CGC_Extended_Application_form_v5_16_May_2014.docx.

DARPA, 2016. Cyber Grand Challenge: Rules Version 3. http://archive.darpa.mil/CyberGrandChallenge CompetitorSite/Files/CGC Rules 18 Nov 14 Version 3.pdf.. URL: http://archive.darpa.mil/CyberGrandChallengeΩCompetitorSite/Files/CGC_Rules_18_Nov_14_Version_3.pdf.

Deng, Z., Xu, D., Zhang, X., Jiang, X., 2012. Introlib: efficient and transparent library call introspection for malware forensics. Digit. Invest. 9, S13–S23.

Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security.

Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R., 2015. Repeatable reverse engineering with panda. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, pp. 4:1–4:11. https://doi.org/10.1145/2843859.2843867. http://doi.acm.org/10.1145/2843859.2843867.

Eagle, C., 2008. The IDA Pro Book: the Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, San Francisco, CA, USA.

Engblom, J., 2012. A review of reverse debugging. In: System, Software, SoC and Silicon Debug Conference (S4D), 2012. IEEE, pp. 1–6.

Hunt, P., Konar, M., Junqueira, F.P., Reed, B., 2010. Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (pp. 11–11). http://dl.acm.org/citation.cfm?id=1855840.1855851.

Intel Corporation, 2015. Dynamic Analysis of Program Behavior Project Contract Number Hr0011-12-9-0002 Final Report.

Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A., 2014. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference.

Rowlingson, R., 2004. A ten step process for forensic readiness. Int. J. Digit. Evid. 2, 1–28.

Shellphish, T., 2017. Cyber Grand Shellphish. Phrack. http://phrack.org/papers/cyber_grand_shellphish.html.

Shields, C., Frieder, O., Maloof, M., 2011. A system for the proactive, continuous, and efficient collection of digital forensic evidence. Digit. Invest. 8, S3–S13.

Vidas, T., Eagle, C., Wright, J., Caswell, B., Thompson, M., Sorenson, H., 2017. Designing and Executing the Worlds First All-computer Hacking Competition: a Panel with the Development Team. Shmoocon. https://archive.org/details/ShmooCon2017.

Willems, C., Holz, T., Freiling, F., 2007. Toward automated dynamic malware analysis using cwsandbox. IEEE Secur. Priv. 5.

Wind River, 2015a. Simics Accelerator User's Guide, 4.8.

Wind River, 2015b. Simics Ethernet Networking, 4.8.

Wind River, 2015c. Simics Hindisght User's Guide, 4.8.