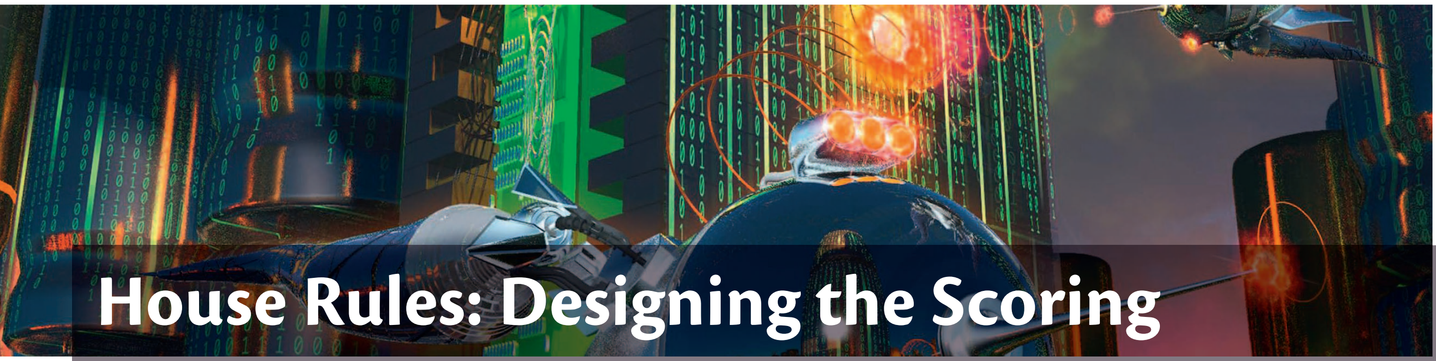
2018-03

House Rules: Designing the Scoring Algorithm for Cyber Grand Challenge

Price, Benjamin; Zhivich, Michael; Thompson, Michael; Eagle, Chris

IEEE

https://hdl.handle.net/10945/57880



House Rules: Designing the Scoring Algorithm for Cyber Grand Challenge

Benjamin Price and Michael Zhivich | MIT Lincoln Laboratory
Michael Thompson and Chris Eagle | Naval Postgraduate School

The key driving force behind any capture-the-flag competition is the scoring algorithm; the Cyber Grand Challenge (CGC) was no different. In this article, we describe design considerations for the CGC events, how these algorithms intended to incentivize competitors, and effects these decisions had on the resulting gameplay.

Scoring algorithms are at the core of any competition. They define the objective of a game and drive competitors' strategies, guiding investments of effort, affecting resulting gameplay, and last (but not least) determining who captures the glory and walks away with the prize. As such, much consideration is given to design of scoring algorithms by organizers of any competition, and the Cyber Grand Challenge (CGC) was no different. CGC's lofty vision was to "engender a new generation of autonomous cyber defense capabilities that combine the speed and scale of automation with reasoning abilities exceeding those of human experts."[1] In particular, CGC challenged competitors with a highly nontrivial task of "improv[ing] and combin[ing] semi-automated technologies into an unmanned Cyber Reasoning System (CRS) that can autonomously reason about novel program flaws, prove the existence of flaws in networked applications, and formulate effective defenses."[1]

In practice, this meant that a CRS would be provided a bespoke, known vulnerable, *challenge binary* that implemented functionality for a never-before-seen network application or service. The CRS would then have to produce a *replacement binary* that mitigated the effects of the embedded vulnerability and a *proof of vulnerability* describing an interaction with the vulnerable binary that would cause it to exhibit undesired behavior (for example, crash, leak sensitive data, and so on).

The infrastructure team organizing this competition arguably had an equally daunting task of creating a sufficiently realistic, yet distinct and protected arena in which these CRSs would compete in the first fully automated attack–defense capture-the-flag (CTF) event. When developing the scoring algorithms that would guide development of the first crop of automated cyber reasoning systems, we focused on ensuring the following properties of scoring:

- *Fairness*. Scoring should not discriminate against a specific method the team uses to solve the problem. Note that artifacts that appear in the solution could be penalized (for example, using too much memory in a replacement binary), but no specific process of bug discovery and remediation should be prescribed or proscribed.
- *Collusion resistance*. We wanted to entice competitors to focus on improvements in automated network defense, not analysis and defeat of the scoring

algorithm. In particular, the scoring algorithm should disincentivize collusion between participants.

- *Real-world relevance*. We wanted the results of this research to produce practical prototypes that could be readily adopted by the industry, so the scoring algorithm aimed to replicate many of the pressures the real world places on security solutions.
- *Automated evaluation*. Because CGC is a machine-scale competition, it would be impossible to score the results by hand if only due to the number of submissions. Thus, scoring cannot rely on "expert judgment" of any kind; all measures required for scoring have to be automated.

Armed with these principles, we considered many different options, and settled on two variants of the same general algorithm—one used for the CGC Qualifying Event (CQE), and one used for the CGC Final Event (CFE); the variant algorithms reflect differences in the number of participants and mechanics of the competition between CQE and CFE.

In the rest of the article, we describe the algorithms themselves, the reasons these formulations were selected, how the algorithms affected gameplay in CQE and CFE, and general lessons learned that might be drawn from our experience to help other CTF competition designers.

Scoring Rubrics: What to Measure?

The first question that comes to mind when creating the scoring algorithm is: What should we measure? Given that the CGC competition is about improving the state of the art in automated network defense, we certainly need some measure of security provided by the replacement binaries produced by competitors' CRSs. However, it is exceedingly easy to provide a binary that has perfect security—one that does nothing. Therefore, we also need to measure the availability of service provided by the replacement binary. Finally, because CGC is a competition focused on creating cyber reasoning systems, the scoring algorithm should reward CRSs that can find a vulnerability in the original binary—we term this rubric *evaluation*.

Security

The world would be a saner place if an analytic technique existed that could examine an application and provide a complete listing of embedded exploitable vulnerabilities; in such a world, we would not need a Cyber Grand Challenge, and we would not suffer from so many cyberattacks. Instead, we have to consider security as a relative, not absolute metric. Attacks provide a concrete demonstration of the vulnerability on which to base measurement.

In CGC, two different entities provided proofs of vulnerability (PoVs), which represented attacks in our game environment: challenge binary (CB) authors and competitors' cyber reasoning systems. Challenge binary authors were required to provide a PoV that demonstrated the exploitability of each vulnerability embedded in a CGC challenge binary. The CRSs of course would also discover and prove vulnerabilities in the challenge binaries that formed the substrate of the competition.

Due to the different provenance of PoVs, two different security scores were created: a *reference security* score that measured how well a replacement binary defended against PoVs provided by CB authors, and a *consensus security* score that measured how well a replacement binary defended against PoVs provided by CRSs.

Availability

As mentioned previously, providing a "perfectly secure" replacement binary would be trivial, if the replacement were not tested for functionality. Clearly, this would violate the guideline to encourage solutions that have real-world applicability, so we needed a way to measure the functionality of the replacement service. To accomplish this task, we relied on CB authors to provide a test case generator that could automatically create thousands of test cases for the application they have developed. Each test case included not only the input to the application under test but also the logic to decide whether the application's response was correct. In effect, the test suite created an automatically checkable specification of the application behavior. A measure of functionality was determined by the number of tests passed by the replacement binary compared to the number of tests passed by the original application.

However, measuring only functionality does not fully satisfy the real-world applicability requirement. In practice, *performance* of an application or service is also of great importance, and many security solutions have not found wide adoption due to their performance overhead. To measure performance impact, we concentrated on three typical performance factors: CPU execution time, memory usage, and file size. The former two metrics were computed in aggregate over the execution of the test cases used as part of the functionality test, whereas the latter involved a simple comparison of file sizes between original and replacement binaries.

Evaluation

Finally, because we wanted to reward creation of cyber reasoning systems, we awarded additional points for finding vulnerabilities. The most indisputable way to prove that a vulnerability exists is to provide an input or interaction with an application that causes the application to exhibit some bad behavior (for instance, crashing or revealing sensitive information). In CGC, this input took a form of an XML-based description (for CQE) or a full binary that interacted with an application over a network socket to prove vulnerability (in CFE).

Scoring Algorithm

The scoring algorithm for CQE and CFE shared the same structure, though different considerations were prioritized when defining each term. The overall algorithm scored each challenge set individually and summed the scores over challenge sets (for CQE) and over both challenge sets and rounds (for CFE). A challenge set score was defined as a product of the scoring rubrics described, because we wanted to signal that all parts of the competition are equally important and excelling at just one aspect would not be sufficient to win:

$$Score(RB, PoV) = Avail(RB) \times Sec(RB) \times Eval(PoV).$$

The *availability* term was defined similarly between CQE and CFE; however, the *security* and *evaluation* terms differed slightly. To explain the different formulations for these score components, let's discuss each event separately.

CGC Qualifying Event

CQE was held halfway through the program as a way to downselect the cyber reasoning systems that would participate in the final, completely automated, attack–defense capture-the-flag competition. Because the event occurred relatively early in the program, it was not conducted as a full-scale attack–defense competition, but instead as an approximation of a single round in the final event. During this event, the cyber reasoning systems were provided a set of 131 previously unseen binaries with embedded vulnerabilities. Some binaries came with a small capture of network traffic that could be used to deduce their intended functionality. The CRS had 24 hours to analyze the binaries, produce replacements that would mitigate embedded vulnerabilities, and provide PoVs (in CQE, this was an input that would cause the vulnerable binary to crash with a segmentation violation or illegal instruction fault). Each entry was then scored according to the above algorithm (summed over all entries to produce the final CQE score), where the terms were defined as described below.

Availability.

$$Avail(RB) = min(Perf(RB), Func(RB))$$

$$Perf(RB) = max\begin{cases} mem\_use\_overhead(RB) \\ cpu\_exec\_time\_overhead(RB) \\ 0.25 \times file\_size\_overhead(RB) \end{cases}$$

$Func(RB) =$ **(Number of successful tests)/(Number of tests)**

As discussed previously, availability measures the performance and functionality of the replacement binary and assigns an availability penalty based on whichever degradation is worse. To combine effects of file size, memory usage, and CPU execution time overhead, a similar approach is taken—the maximum of the three performance overheads is selected. In addition, both *Perf* and *Func* scores are passed through functions that provide a "faster-than-linear" decrease in score (see Figure 1a and Figure 1b, respectively). The idea behind these curves was to provide better differentiation between higher-performing teams and ensuring that a smooth transition is made from maximum availability score of 1 to minimum of 0. Note that the *Perf* graph provides a 10 percent "grace factor"—that is, if the performance of the replacement binary is within 10 percent of the performance of original binary, then no performance penalty is applied. At the other end, a performance overhead of 100 percent or more results in 0 availability score.

Security.

$$Sec(RB) = \begin{cases} if\ SecRef(RB) > 0: \\ 1 + \frac{1}{2} \times (SecRef(RB) + SecCon(RB)) \\ otherwise: 0 \end{cases}$$

In CQE, security was evaluated both against the PoVs provided by CB authors (*SecRef* for reference) and CRS-provided PoVs (*SecCon* for consensus). The former provided an unbiased measure of whether the CRS has mitigated exploitability of a bug inserted by a challenge binary author; at least one PoV in this category had to be mitigated by the replacement binary in order for security score to be non-zero. Because other PoVs for the same bug might exist, the competitor-provided PoVs were also tested against all replacement binaries for that challenge set. If a replacement binary mitigated all competitor PoVs, its *SecCon* score was set to 1; otherwise, it was 0.

The reasons for this formulation of the security score derive from the requirement that the scoring algorithm be collusion resistant. Because CQE was conducted online and open to a large number of participants, we were concerned that a team might register many "sock puppet" teams that would submit PoVs that the "master" team would know how to defend, thus artificially inflating its security score while providing no additional security. Therefore, we could not give equal weighting to reference PoVs and consensus PoVs. A detailed description of the red-teaming exercise that led to this decision can be found in the CGC FAQ.[2]

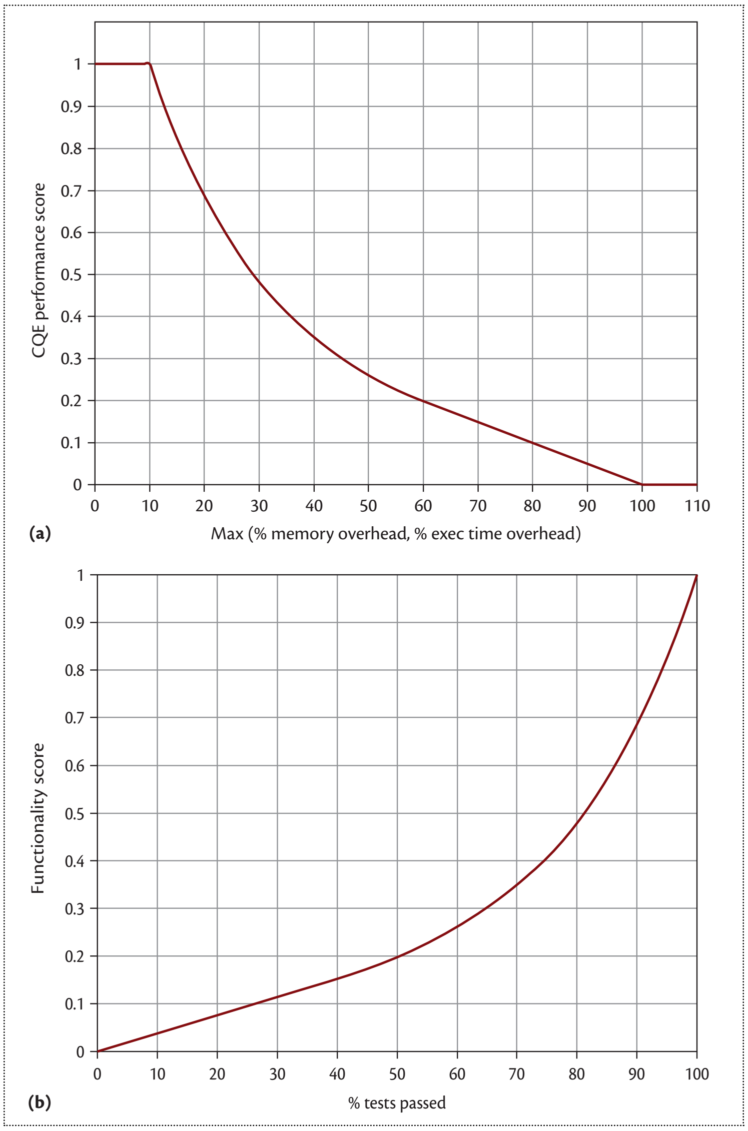


Figure 1. Curves illustrating conversions for (a) performance and (b) functionality.

Evaluation.

$$Eval(PoV) = \begin{cases} 2 & Submitted\_PoV\_successful \\ 1 & otherwise \end{cases}$$

The evaluation portion of the score was straightforward: teams were awarded a 2x multiplier for providing a successful PoV against a reference challenge binary, thus increasing that team's score. Note that providing a working PoV may also decrease a competitor's score by removing their SecCon bonus.

Discussion. Putting all the terms together, we have the following possible score values:

- *Balanced CRS.* A CRS that solves the challenge completely with perfect retention of functionality and performance (and a successful PoV) would receive a score of $1 \times 2 \times 2 = 4$.
- *Defense-only CRS.* A CRS that uses a defense-only strategy with perfect retention of functionality and performance would receive a score of $1 \times 2 \times 1 = 2$.
- *Offense-only CRS.* A CRS that uses an offense-only strategy would receive a score of $1 \times 0 \times 2 = 0$.
- *Do-nothing CRS.* A CRS that just returned the original vulnerable binary would receive a score of $1 \times 0 \times 1 = 0$.

The general idea behind the CQE scoring algorithm is that the teams that could automatically mitigate PoVs while maintaining functionality and performance of an application should receive a high score. If they can also provide a PoV against the original binary, their score is increased and their competitors' consensus scores might decrease. We wanted to reward finding vulnerabilities, but not preclude defense-only solutions. Hence, the evaluation factor will not cause the score to be 0; however, a purely offense-oriented solution was deemed insufficient, so a team could not score points by just providing a PoV, without associated nontrivial defense in the replacement binary. Thus, we were selecting teams with both good defense and good offense to advance to the finals.

CGC Final Event

CGC Final Event was a completely autonomous attack–defense CTF among the seven finalist CRSs determined by total CQE scores. The structure of CFE differed significantly from CQE—it consisted of 96 rounds during which new challenge binaries could be introduced into the game or old ones retired. Each challenge binary fielded by a CRS was evaluated for functionality and performance, with feedback provided to the CRS. Each CRS had an opportunity to download replacement binary and network filter submissions made by other competitors during the previous round and evaluate them to find any new or remaining vulnerabilities. Each CRS could also submit a PoV to be launched against a specific competitor and receive feedback of its success or failure.

There were two different kinds of PoVs in CFE: Type 1 (controlled crash) and Type 2 (memory disclosure), which modeled different attack types. Type 1 PoV required that the vulnerable binary would crash with specific values of instruction pointer and a general-purpose register that were prenegotiated with the game infrastructure—this was a proxy for *remote code execution* attack. Type 2 PoV required that an interaction with the vulnerable binary result in leaking 4 consecutive bytes from the so-called magic page that was mapped at a fixed address and contained specially crafted pseudorandom data that changed each time the binary started—this was a proxy for a *secret stealing* attack. A more detailed description of CFE PoVs can be found in "Proof of Vulnerability Markup Language."[3]

The overall structure of the scoring algorithm remained the same as in CQE (the scores were now summed over both challenge sets and rounds to arrive at the final score); however, several of the terms required adjustments to capture the new complications of rounds, consensus evaluation of security, and introduction of network-based defenses.

Availability.

$$Perf(RB) = \max \begin{cases} mem\_use\_overhead(RB) \\ cpu\_exec\_time\_overhead(RB) \\ wall\_time\_overhead(RB) \\ 0.25 \times file\_size\_overhead(RB) \end{cases}$$

The performance portion of the availability term was slightly adjusted to include a new component of wall-clock time in addition to existing CPU execution time. This new component reflected the measurement of slowdown incurred by using a network-based filter to detect or respond to attacks; the wall time was measured at the host performing the functionality test. In addition, the grace factor was lowered from 10 percent to 5 percent to correspond more directly to existing industry requirements for readily adoptable security solutions.[4]

Furthermore, to enable consensus evaluation of replacement binaries, CFE infrastructure imposed a one-round delay on running the replacement binary or associated network filters. This meant that a team replacing a binary would get an availability score of 0 for the service being replaced in that round, while the other competitors would get an opportunity to analyze the binary for new or remaining vulnerabilities. This dynamic was intended to emulate real-world practices where downtime is very costly and rarely tolerated.

Security.

$$Sec(PoV) = \begin{cases} 1 & any\_competitor\_PoV\_successful \\ 2 & otherwise \end{cases}$$

The security term changed as well. Because we expected the replacement binaries to change in response to attacks from other competitors, it no longer made sense to evaluate security using reference PoVs provided by CB authors as they were unlikely to work against the replacement binaries after the first round. Therefore, this score component reflected a more empirical notion of security: if any competitor successfully proved vulnerability in the replacement binary, the score was set to 1; otherwise, it was set to 2.

Evaluation.

$Eval(PoV) = 1 + \textbf{(Number of successful PoVs)/(Number of competitors} - \textbf{1)}$

The evaluation term was modified as well, because now a CRS could score against up to six competitors in a single round. Note that a CRS could submit a different PoV against each competitor to target that competitor's replacement binary.

Discussion. It is more difficult to describe possible scoring values in an adversarial environment because security and evaluation scores now depend heavily on the actions of the competitors. For the sake of example, let's suppose there are only two teams in CFE, where Team A plays a specific strategy and Team B does nothing. In that case, we have the following possible score values:

- *Balanced CRS.* If Team A's CRS solves the challenge completely with perfect retention of functionality and performance (and a successful PoV), it would receive a score of $1 \times 2 \times 2 = 4$, while Team B would receive a score of $1 \times 1 \times 1 = 1$.
- *Defense-only CRS.* If Team A's CRS uses a defense-only strategy with perfect retention of functionality and performance, it would receive a score of $1 \times 2 \times 1 = 2$, while Team B would receive a score of $1 \times 2 \times 1 = 2$.
- *Offense-only CRS.* If Team A's CRS uses an offense-only strategy, it would receive a score of $1 \times 2 \times 2 = 4$, while Team B would receive a score of $1 \times 1 \times 1 = 1$.
- *Do-nothing CRS.* If Team A's CRS did nothing, it would receive a score of $1 \times 2 \times 1 = 2$, while Team B would also receive the same score.

The effect of these changes made CFE scores rather different from CQE: the security score would no longer cause a competitor to receive a 0 for a round; in fact, doing nothing (that is, neither replacing a binary nor providing PoVs against competitors) guaranteed a score of 1 if a service was successfully attacked and a score of 2 if no attacks against it were successful. Note that the dynamics between CRSs become much more important in this game—an offense-only strategy works only if the competitor is not expected to find the PoV in the network traffic and turn it around to attack its creator, thus leveling the score.

Competitor Strategies

Given that the intention of the scoring algorithm design was to encourage finding and understanding bugs (by generating PoVs) and automatically patching vulnerabilities without excessive performance or functionality degradation, what strategies did they select? Were there unintended consequences? In this section, we tackle some of these questions and discuss competitors' strategies in CQE and CFE.

Wall-Time Effects on Competitor Choices

The use of wall time to assess the performance impact of a network filter seemed to affect competitor behavior in at least two ways. First, most competitors avoided the use of network filters entirely; the few filters that were deployed were very simple—they terminated sessions that contained data resembling references to the magic page in attempts to foil Type 2 PoVs. Post-event interviews suggested some teams lacked confidence that they could estimate the performance impact of fielding network filters, and chose to forgo this capability in fear of heavy performance penalties. This finding ran counter to our expectation of teams trying to offload computation into the network appliance, which motivated measuring its performance impact in the first place.

The second behavior that was likely prompted by the use of wall time in the scoring algorithm was the inclusion of infinite loops by one of the teams in their PoVs to degrade the performance of their competitors' services. This team would create PoVs that used remote code execution to enter a tight infinite loop (equivalent of `while (1) ;`) subsequent to scoring (for example, by leaking values of the memory page). This consumes defended host CPU resources and increases the response time of other services on the defended host. We did not perform analysis to determine if the use of this "Type 3 PoV" had a measurable impact on the scores of their competitors. But the fact that at least two teams managed to significantly degrade the performance of their own services by fielding flawed replacement binaries suggests that this strategy has merit.

In CGC, we specifically worked to remove the ability of competitors to cause denials of service by flooding hosted services with traffic. Each CRS could only launch a limited number of PoVs against each service on each competitor's defended host. The Type 3 PoV turned out to be a clever hack to bring down a competitor's score in addition to improving your own; the one downside to this phenomenon was that availability scores for services unrelated to the one being exploited could suffer, which made the process of identifying the culprit for poor performance difficult.

Always Be Throwing

The scoring algorithm imposes no penalties on a team for deploying PoVs against their competitors. Teams could throw PoVs up to 10 times per round against each of their competitors' services; a few teams elected to throw generic PoVs whenever they lacked a working, targeted PoV. One team's generic PoVs simply guessed at a magic page value at random (on one occasion the guess was correct!). However, this team's logic for fielding generic PoVs may have been flawed, because on at least two occasions, their CRS replaced successful targeted PoVs with generic PoVs, which failed to score in subsequent rounds.

From a game-theoretic perspective, there was no disincentive to throw PoVs, and launching attacks (even nonworking ones) provided teams with ability to create additional traffic that would have to be filtered by the network appliance and analyzed by the target team. In addition to using up limited resources, this traffic could be used as a cover for real PoV throws, making them harder to identify.

Zero Score for Consensus Round

The cost of replacing a service in CFE was intended to dissuade teams from relying on frequent replacement as a defensive strategy. Post-event interviews with the teams suggested that this cost led some teams to postpone the submission of replacement binaries and to avoid replacing a binary more than once.

At one extreme, Team A generally only replaced a binary following indication that the associated service was compromised. Their CRS would then not replace that same service again, despite feedback indicating the replacement had zero availability due to performance or functionality degradation. This seemed to be a bug—a post-event interview with Team A indicated that they removed logic that responded to poor availability at the last minute.

Team B employed a different strategy to avoid multiple replacements; their CRS replaced all but three of the services, seemingly independent of whether or not a successful PoV was launched against their services. Team B would never replace that service again, unless the CRS received feedback that the replacement binary had poor availability, in which case the CRS would revert to the original binary. In some instances, Team B's CRS incorrectly reverted services with perfect availability due to a self-inflicted denial-of-service problem caused by heavy CPU usage by an unrelated service running on the shared defended host.

Team C followed this general strategy as well, though they only replaced half of their services, regardless of whether the services had been compromised. Whenever Team C's CRS received feedback that a service had poor availability, it generally uploaded a new replacement binary in response.

From a game-theoretic perspective, the strategies chosen by these teams seem suboptimal. Ideally, a CRS should have enough situational awareness to determine when a successful PoV against their service has been launched and replace the binary with a patched version at that point. Preemptive replacement of services unnecessarily cost several teams points due to the consensus round downtime, even though no successful PoVs were fielded against these services in CFE.

Defenses in CQE versus CFE

During CQE, teams knew that a working PoV would be deployed against each of their replacement binaries, which motivated the deployment of defenses—in fact, if no reference PoVs were mitigated, a team would receive a score of 0 on that submission. In CFE, reference PoVs were not deployed against the services, and any successful attacks would have to come from competitors. Thus, a team's decision to deploy a defense might depend on whether they believed a given service could be compromised by one of their competitors.

In a PoV-rich environment, where many services are proven vulnerable, a good strategy (one achieving the highest score) might be to focus on patching and patch preemptively, as it might prevent a round of reduced score when the service is proven vulnerable. However, in a PoV-limited environment, where many services are not proven vulnerable, a good strategy would be to patch only in response to a successful attack, as discussed earlier. In CFE, only 20 challenge sets out of 82 were proven vulnerable by competitors, so a strategy of patching in response to an attack would have produced higher scores. The use of reference PoVs in CFE might have provided additional motivation to field defenses and enabled better demonstration of CRS patching capabilities observed in CQE.

Point Patches versus Generic Patches

While the design of the competition was not intended to preclude the use of any given defensive strategy, we did strive to encourage correcting the program flaws rather than providing generic mitigations that simply masked the presence of those flaws. We did not feel that liberal application of control flow integrity techniques would advance the state of the art of network defense. The availability element of the scoring algorithm was designed in part to reward targeted patches by penalizing increased memory use and CPU execution time.

However, as can been seen in Figure 2, many replacement binaries were fielded that consumed significantly fewer resources than the associated reference services. Figure 2 shows ratios of resources consumed by replacement binaries divided by resources consumed by reference binaries, averaged over each round. Each color in the figure represents a different team, and the different services are arrayed along the *x*-axis.

The ability of a replacement binary to consume fewer resources than the reference service is an unintended effect resulting from the unforgiving performance scoring function combined with the fact that CGC challenge binaries were simple services and thus much smaller than most real-world network services. Some utilized a very small number of pages of memory, and thus increased memory usage of even a single page might drive the availability score to zero. CPU execution times of very short-running services presented a similar problem because any modest increase in processing time resulted in large proportional increases in the measurement of CPU cycles.

To mitigate this problem of very small challenge binaries, the CGC build process deliberately included extra data (in the form of an embedded PDF) and processing (in the form of a CRC computation across the PDF). Neither extra data nor extra computation was necessary for correct service functionality; therefore, both could be safely removed from each binary. During the testing phase before CFE, it was possible for teams to learn that space and time cushions could be obtained by removing the CRC-related code and the PDF data. As a result, teams were able to deploy generic defenses without incurring significant availability costs. This experience indicates that availability constraints alone cannot effectively mandate the use "point patches" rather than generic defenses on small programs.

Effect of Consensus Evaluation

One of the motivations for consensus evaluation was to afford teams the opportunity to find bugs inadvertently introduced by competitors' patches. We found no evidence of any team successfully exploiting a vulnerability unintentionally introduced by another team. We did note two instances of intended insertions of vulnerabilities, motivated by the knowledge that competitors could analyze patched binaries. The first insertion was a honeypot that one team included in most of their replacement binaries. This honeypot was a simple buffer overflow that was easy to find and exploit, but could not be reached when the service was executing on CGC Final Event hardware as a result of an execution divergence between CGC hardware and common analysis environments due to handling of `cpuid` instruction. This honeypot caused several opposing teams to field PoVs designed to exploit the unreachable flaw, thus providing an effective security countermeasure.

The second type of deliberate flaw was a back door embedded in a replacement binary, intended to be

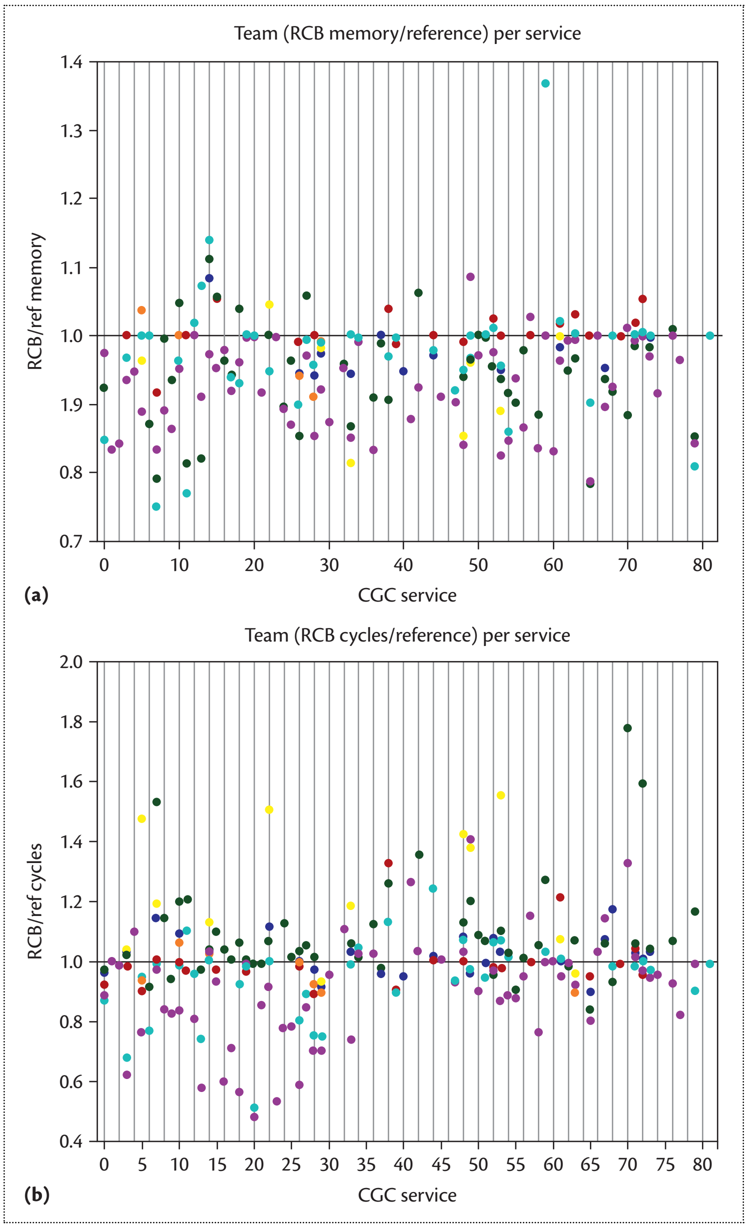


Figure 2. Average (a) memory use and (b) execution time factors for CFE replacement binaries. Each point represents average memory use or CPU execution time factor for replacement binaries by a particular CRS for a particular service.

exploited if another team elected to utilize this binary as their own replacement for the backdoored service. Post-event interviews indicated that multiple teams inserted back doors into their replacement binaries; however, no backdoored service was redeployed during CFE. This behavior was anticipated by CGC organizers; earlier in the project, some teams expressed concerns that their replacement binaries might get reused by the competitors who would be effectively "free-riding"; adding a back door accessible only to the team that created the patch dissuades such behavior.

Lessons Learned

Designing and running any capture-the-flag event is a nontrivial undertaking; organizing a high-profile completely automated capture-the-flag event is doubly so. Despite best efforts on the part of the game designers, some things do not go according to plan, and unintended consequences of scoring or measurement decisions can drive competitors to nonoptimal strategies. In this section, we review several lessons learned as part of organizing and running the CGC Qualifying and Final Events.

Red-Team Scoring Algorithm

When designing the scoring algorithms for CQE and CFE, we spent much time considering competitor strategies that would achieve good scores but not advance the state of the art in automated network defense. In several cases, we had to revise the scoring algorithm to provide disincentives for such strategies (for example, requiring that at least one reference PoV is mitigated in CQE or assigning a significant penalty for service replacement). Much of this effort also focused on discouraging collusion between teams. When performing such analyses, it is useful to consider teams that might "do nothing," teams that might play defense only, offense only, or some sort of balanced or randomized strategy. Each team persona might illuminate a different corner of the scoring space and provide ideas for improving the scoring and redirecting competitors away from undesirable strategies.

Make Measurements Reproducible

Any scoring algorithm is closely tied to the measurements that support it; in case of CGC, the measurements included functionality, wall time, CPU execution time, memory usage, and whether a PoV thrown against a challenge binary successfully proved vulnerability. When designing the measurement framework and scoring algorithm, we focused on ensuring reproducibility of scores; that is, running the same round multiple times with the same inputs should produce the same scores for the competing teams. This meant that the game infrastructure had to go through great pains to control the use of randomness in the game (all randomness available to the challenge binaries and PoVs was an output of a pseudorandom number generator with a known seed) and limit effects of other nondeterminism sources: process scheduling, networking issues, and so on. When a particular measurement could not be contained to an acceptable level of variance due to these sources of nondeterminism, the scoring algorithm had to be adjusted to contain the effects from such measurements on the resulting score.

Make Metrics Transparent

Much of the scoring algorithm design for CGC was driven by the maxim that "you get what you measure." By measuring additional memory usage and additional CPU cycles, we hoped to get patches that focused on the flaw rather than general program-hardening techniques. However, this strategy really only works if the competitors understand what exactly is being measured. Instead of providing a clear statement that we would measure CPU cycles consumed while the process executed in user space, we provided a series of oracles that the teams could interact with to divine the effects of techniques embedded in their replacement binaries on availability. Prior to CQE, this was a sequence of "scored events," and for CFE, this was the "sparring partner." Part of the reasoning for this choice was to prevent the competitors from gaming the scoring mechanism; however, it resulted in too much ambiguity and caused some competitors to model the performance metrics incorrectly.

Avoid Wall Time as a Metric

Our choice to use wall time as a metric to measure availability costs of network filters substantially complicated implementation and testing of the CGC game infrastructure. One problem is that services that complete relatively quickly become very sensitive to small variations in wall time. Thus, differences in kernel scheduling or TCP networking effects could result in significant variations between otherwise identical sessions. This complicated our ability to achieve repeatable results, which are necessary when establishing a baseline against which to measure the performance of replacement binaries and network filters.

We investigated an alternative to wall time by measuring aggregate CPU cycles on the network appliance component for all polls of a given service. However, when first attempted, the network appliance was hosted on Linux, and the CPU cycle measurements had very high variations. We were finally able to achieve repeatable wall-time measurements after converting all of the infrastructure components to FreeBSD and carefully tuning the kernel configurations.

In this article, we presented our experience and lessons learned in designing and implementing the scoring algorithms for the CGC Qualifying and Final Events. These algorithms succeeded in incentivizing competitors to develop systems that could automatically patch previously unseen binaries to mitigate vulnerabilities as well as provide proofs of vulnerabilities for these binaries. The scoring algorithms in CGC were designed for automated evaluation and strived to achieve fairness, collusion resistance, and real-world relevance, and in many aspects we believe that they succeeded. We hope that knowledge of our experience proves useful to other capture-the-flag competition designers and helps them avoid some of the pitfalls we faced. ■

Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US government.

References

1. "Cyber Grand Challenge: Rules," Version 3, DARPA; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_Rules_18_Nov_14_Version_3.pdf.
2. "Cyber Grand Challenge: Frequently Asked Questions (FAQ)," DARPA; http://archive.darpa.mil/CyberGrandChallenge_CompetitorSite/Files/CGC_FAQ.pdf.
3. "Proof of Vulnerability Markup Language," DARPA; https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/cfe-pov-markup-spec.txt.
4. "The BlueHat Prize Contest Official Rules," Microsoft; https://web.archive.org/web/20111120054734/http://www.microsoft.com:80/security/bluehatprize/rules.aspx.

Benjamin Price is a member of the Cyber Analytics and Decision Systems Group at MIT Lincoln Laboratory. Email at ben.price@ll.mit.edu.

Michael Zhivich joined MIT Lincoln Laboratory in 2005 as a member of the Secure Resilient Systems and Technology Group. Email at mzhivich@gmail.com.

Michael Thompson is a research associate at the Naval Postgraduate School in Monterey, California. Email at mfthomps@nps.edu.

Chris Eagle is a senior lecturer of computer science at the Naval Postgraduate School in Monterey, California. Email at cseagle@nps.edu.