INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 666-2900 • FAX (510) 666-2956

GQ: Realizing a System to Catch Worms in a Quarter Million Places

Weidong Cui[†], Vern Paxson[‡], and Nicholas C. Weaver[‡]
[†]University of California, Berkeley, CA
[‡]International Computer Science Institute, Berkeley, CA

TR-06-004

September 7, 2006

Abstract

A key tool for detecting new worm outbreaks in their early stages is the *honeyfarm*, a large collection of honeypots fed Internet traffic by a "network telescope". However, actual operation of a honeyfarm in a large-scale environment presents difficult scaling challenges. We discuss the structure and implementation of *GQ*, a honeyfarm system we built to analyze in real-time the scanning probes seen on a quarter million Internet addresses. GQ's architecture emphasizes high fidelity, scalability, isolation, stringent control, and wide coverage. We examine how the implementation endeavors to achieve each of these, evaluate its effectiveness in doing so, and report on preliminary experiences with operating the system at scale, during which we captured 66 distinct worms over the course of four months.

1 Introduction

The combination of widespread software homogeneity and the Internet's unrestricted communication model has created an ideal climate for infectious pathogens, one that in recent years has seen ongoing malware innovation in terms of increasing speed, virulence, and sophistication. A central tool that has emerged recently for detecting Internet outbreaks is the *honeyfarm*, a large collection of honeypots fed Internet traffic by a "network telescope" [1, 10, 18, 25]. The honeypots interact with remote sources whose probes strike the telescope, using either "low fidelity" mimicry in order to characterize the *intent* of the probe activity [1, 20, 21, 30], or with "high fidelity" full execution of incoming service requests in order to purposefully become infected as a means of assessing the malicious activity in detail [5, 9, 10, 25].

Considerable strides have been made in deploying low-fidelity honeyfarms at large scale [1, 30], but high-fidelity systems face extensive challenges, due to the much greater processing required by the honeypot systems, and the need to safely contain hostile, captured contagion while still allowing it enough freedom of execution to fully reveal its operation. Recent work has seen significant advances in some facets of building large-scale, high-interaction honeyfarms, such as making much more efficient use of Virtual Machine (VM) technology [25], but these systems have yet to see live operation *at scale*.

In this work we present our realization of a high-fidelity honeyfarm system that operates on attacker probes captured by a network telescope whose reach exceeds 250,000 Internet addresses. The system, named *GQ*, aims to enable the automatic detection of incipient network epidemics, and, by monitoring their operation, the extraction of the actionable information necessary to control defense mechanisms.

We discuss the structure and implementation of *GQ* and preliminary experiences with operating it at scale. GQ's architecture emphasizes (i) high fidelity, (ii) scalability, (iii) isolation, (iv) stringent control, and (v) wide coverage. To achieve fidelity, GQ runs fully functional servers across a range of operating systems known to be prime targets for worms (especially various flavors of Microsoft Windows). We obtain scalability through aggressive filtering, including application-independent "replay", a technology for reproducing one side of a previously seen application dialog without requiring semantic knowledge of the application's protocol [4]. Using replay, we can detect and filter out instances of previously seen malcode after a potentially intensive—but scripted and thus lightweight—dialog. We isolate honeypots on their hosts using VM technology (currently VMware, but with an architecture readily adaptable to the Xen-based system developed by our colleagues [25]), and from other honeypot platforms using VLANs. For control, the system includes dynamic redirection, a containment policy enforcer, and an independent "policer" to serve as an safety net. Finally, we have built GQ to support a wide variety of honeypot images and types.

Of particular note for GQ is our extensive use of filtering mechanisms to winnow down the hundreds of probes/sec captured by the front-end telescope to a level that our VM systems can process. Along with the commonly used approach of prefiltering the traffic to remove repeated scans from the same source, GQ's architecture also includes a second tier of filtering that can eliminate known attacks even when the attack has a complex, multi-connection structure. We pursue this second filtering mechanism using an extension of Cui's RolePlayer system, an application-independent "replay" mechanism [4]. Given examples of an application session (such as attacks), RolePlayer can extract a "script" and then follow it to mimic both the client side and the server side of the session for a wide variety of application protocols, *without* requiring knowledge of any specifics about the particular application.

This filter element interacts with remote sources by navigating through a series of "scripts" it has automatically derived from previously seen attacks. If the source follows a script, the filter can classify it as a known attack (generally discarding the session at that point). However, if the source deviates from the script, we then use the replay capability in reverse to re-run the traffic—originally sent by the source to the filter—against a live honeypot server to bring it in synch with the source. At this point, the source interacts with the full server, allowing us to observe the effects of the previously unseen activity. If the attack turns out to be of interest, we can then also use replay to launch it against additional system configurations as a means to automatically derive a "toxicology spread" indicating the vulnerability profile of the attack; or to test whether a defense derived from analyzing the attack does indeed block subsequent attacks.

The next section gives an overview of related work. We discuss GQ's design and implementation in §3

and §4, and report on our operational experiences in §5, including an evaluation of the efficacy of GQ's different elements. We offer thoughts on future work in §6 and conclude in §7.

2 Related Work

Understanding the behavior of Internet-scale worms requires broad visibility into their workings. *Network telescopes*, which work by monitoring traffic sent to unallocated portions of the IP address space, have emerged as a powerful tool for this purpose [18], enabling analysis of remote denial-of-service flooding attacks [19], botnet probing [29], and extensive analysis of worm outbreaks [12, 15, 16, 17].

However, the passive nature of network telescopes limits the richness of analysis we can perform with them; often, all we can tell of a source is that it is probing for a particular type of server, but not what it will do if it finds such a server. We can gain much richer information by interacting with the sources. One line of research in this regard has been the use of lightweight mechanisms that establish connections with remote sources but process the data received from them syntactically rather than with full semantics. For example, *iSink* uses a stateless active responder to generate response packets to incoming traffic [30], enabling it to discriminate between different types of attacks by checking the response payloads. The lightweight responder of the *Internet Motion Sensor* [1] bases its analysis on payload signatures. The responder acknowledges TCP SYN packets to elicit the first data segment the source will send, using a cache of packet payload checksums to detect matches to previously seen activity. This matching, however, lacks sufficient power to identify new probing that deviates from previously seen activity only in subsequent data packets, nor can it detect activity that is semantically equivalent to previous activity but differs in checksum due to the presence of message fields that do not affect the semantics of the probe (e.g., IP addresses embedded in data). Pang *et al* used a detailed application-level responder to study the characteristics of Internet "background radiation" [20]. The work shows that a large proportion of such probing cannot in fact be distinguished using lightweight first-packet techniques such as those in [1, 30] due to the prevalence of protocols (such as Windows NetBIOS and CIFS) that engage in extensive setup exchanges.

A *honeypot* is a vulnerable network decoy whose value lies in being probed, attacked, or compromised for the purposes of detecting the presence, techniques, and motivations of an attacker [23]. Honeypots can run directly on a host machine ("bare metal") or within virtual machine environments that provide isolation and control over the execution of processes within the honeypot. Honeypots range from "low interaction" (no actual execution of potentially vulnerable servers) to "high interaction" (true, full execution); see [25] for a concise overview.

Advances in Virtual Machine Monitors such as VMware [24], Xen [2], and User-Mode Linux (UML) [6], have made it tractable to deploy and manage high-interaction virtual honeypots. The Honeynet project [9] proposed an architecture for high-interaction honeypots used to contain and analyze attackers in the wild, focusing on two main components: data control and data capture. The data control component prevents attackers from using the Honeynet to attack or harm other systems. It limits outbound connections based on configurable policies and uses *Snort Inline* [14] to block known attacks. The data capture component logs all of the attacker's activity. It uses a hidden kernel module to log and transmit data to a safe machine. In [5], Dagon and colleagues present HoneyStat, a system that uses high-interaction honeypots running on VMware GSX server to monitor memory, disk and network activity on the guest OS. HoneyStat uses logit regression for causality analysis to detect a worm when discovering that a single network event causes all subsequent memory or disk events.

By themselves, honeypots serve as poor "early warning" detection of new worm outbreaks because of the low probability that any particular honeypot will be infected early in a worm's growth. Conceptually, one might scatter a large number of well-monitored honeypots across the Internet to address this limitation, but this raises major issues of management, cost, and liability. Spitzner presented the idea of "Honeypot Farms" to address these issues [22]. These work by deploying a collection of honeypots in a single, centralized location, where they receive suspicious traffic redirected from distributed network locations. Jiang and Xu present a VM-based honeyfarm system, *Collapsar*, which uses UML to capture and forward packets via

Generic Routing Encapsulation (GRE) tunnels and *Snort-Inline* to contain outbound traffic. However, since in Collapsar each honeypot has a single IP address, the probability of early detection is still low due to the limited "cross section" presented by the honeypots.

Our approach combines clusters of high-interaction honeypots with the large address spaces managed by network telescopes. By leveraging extensive filtering and engaging honeypots dynamically, we can use a small number of honeypots to inspect untrusted traffic sent to a large number of IP addresses, with goals of automatically detecting Internet worm outbreaks within seconds, and facilitating a wide variety of analyses on captured worms, such as extracting (vulnerability, attack, and behavioral) signatures, and testing whether these signatures can correctly detect repeated attacks by deploying them within the controlled environment of the honeyfarm. Architecturally, this is similar in spirit to UCSD's work on the *Potemkin* system, though that effort particularly emphasizes developing powerful VM technology for running large numbers of servers optimized for worm detection [25].

Finally, we make extensive use of Cui's *protocol-independent replay* [4]. This work developed the *RolePlayer* system which, given examples of an application session, can mimic both the client side and the server side of the session for a wide variety of application protocols. A key property of RolePlayer is that it operates in an *application-independent* fashion: the system does not require *any* specifics about the particular application it mimics, other than incorporating a few low-level syntactic conventions such as representing IP addresses using "dotted quads", and contextual information such as the domain names of the participating hosts. RolePlayer's application independence is of great benefit to GQ, as our #1 goal is to detect outbreaks reflecting attacks never seen before. We extend it significantly by broadening its mechanism to follow multiple possible dialogs in parallel.

Concurrent with the RolePlayer work, Leita *et al* proposed ScriptGen [13], a system for automatically generating *honeyd* [21] scripts without requiring prior knowledge of application protocols. They present a novel region analysis algorithm to identify and merge fields in an application protocol. While these results hold promise for a number of applications, it remains a challenge to use the approach for completing interactions with attacks using complex protocols such as SMB.

3 GQ Design

In this section we present GQ's design, beginning with an overview of the system's architecture. We then discuss management of the incoming traffic stream: obtaining input from the network telescope, prefiltering this stream with lightweight mechanisms, further filtering it using a "replay proxy". We follow this with how we address the thorny problem of "containment," detailing the containment and redirection policy we have implemented that endeavors to strike a tenable balance between allowing contagion executing within the honeyfarm to exhibit its complete infection (and possibly self-propagation) process, while preventing it from damaging external hosts. We finish with a discussion of issues regarding the management of the honeypots.

3.1 Overview and Architecture

As Figure 1 portrays, GQ's design emphasizes high-level modularity, consisting of a front-end controller and back-end honeypots, mediated by a honeypot manager. This structure allows the bulk of our system to remain honeypot-independent: although we currently use VMware ESX server for our honeypot substrate, we intend to extend the system to include both Xen-based honeypots using the technology developed for Potemkin [25] and a set of "bare metal" honeypots that execute directly on hardware rather than in a VM environment, so we can capture malware that suppresses its behavior when operating inside a VM.

The bulk of GQ is written in Click [11], a C++ framework for building packet processors. The front-end controller supports data collection from multiple sources, including both direct network connections and GRE tunnels. The latter allow us to topologically separate the honeyfarm from the network telescope, and potentially to scatter numerous "wormholes" around the Internet to obtain diverse sensor deployment. These features are desirable both for masking the honeyfarm's scope from attacker discovery and to ensure

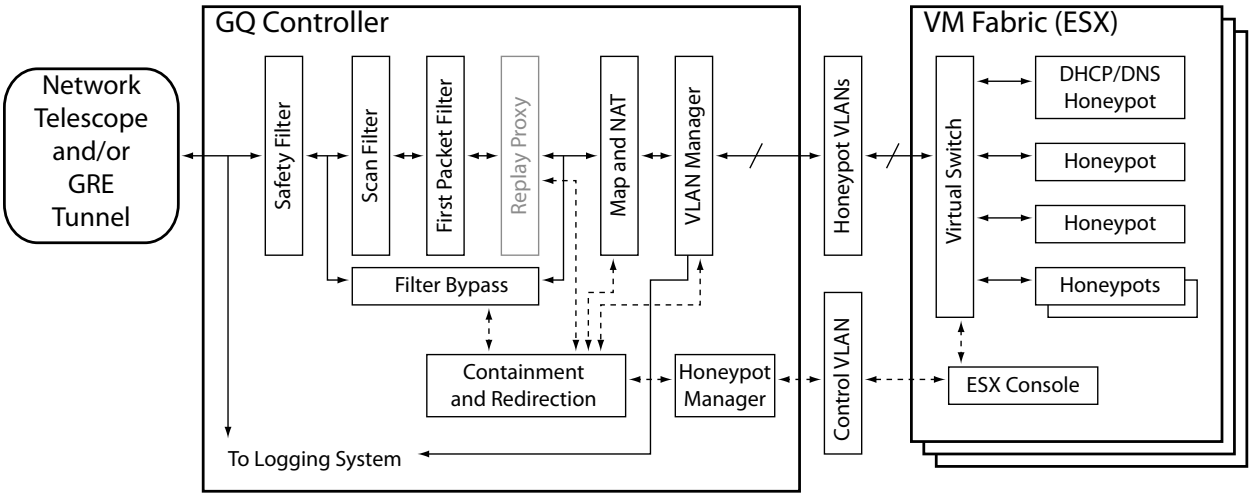


Figure 1: GQ's general architecture

visibility into a broad spectrum of the Internet address space, as previous work has shown large variations in the probing seen at different locations in the network [3]. The controller also performs extensive filtering, using both simple mechanisms and the more sophisticated replay proxy.

The front-end also performs Network Address Translation (NAT). Rather than require that each honeypot be dynamically reconfigured to have an internal address equivalent to the external telescope address for which it is responding, we use NAT to perform this rewriting. However, if honeypot itself supports dynamic address reconfiguration (e.g., [25]), the front end simply skips the NAT.

The final function of the front-end is containment and redirection. When a honeypot makes an outgoing request, GQ must decide whether to allow it out to the public Internet. Based on the policy in place, the front-end either allows the connection to proceed, redirects it to another honeypot, or blocks it. Redirection to another honeypot is always of interest, as this is the means by which we can directly detect a worm's self-propagating behavior (namely if, once contacted, the new target honeypot itself then initiates network activity). However, many types of malware require multi-stage infection processes for which the initial infection will fail to fully establish (and thus fail to exhibit subsequent self-propagation) if we do not allow some preliminary connections out to the Internet. Finding the right balance between these two takes careful consideration, as we discuss below.

3.2 Network Input

The initial input to GQ is a stream of network connection attempts. GQ can process two different forms of input: a directly routed address range, and GRE-encapsulated tunnels. A direct-routed range simply is a local Ethernet or other network device connected to one or more subnets, while a GRE tunnel requires a remote peer. In both cases, the front-end maintains a mapping of all external↔internal address pairs.

Supporting direct connections gives a simple and efficient deployment option. Supporting GRE gives the opportunity to decouple the honeyfarm system from the telescope's address space, and can ultimately serve as a bridge between multiple honeyfarms.

3.3 Filtering

We term the initial stage of responding as *prefiltering*, which aims to use a few simple rules to eliminate probe sources likely of low interest. Prefiltering has several components. First, GQ limits the total number of distinct telescope addresses that engage a given remote source to a small number N. The assumption here

is that a modest N suffices to identify a source engaged in a new or otherwise interesting form of activity. This filter can greatly reduce the volume of traffic processed by the honeyfarm because of the very common phenomenon of a single remote source engaging in heavy scanning of the telescope's address space.

We note that there is no optimal value for N. Too low, and we will miss processing sources that engage in diverse scanning patterns that send different types of probes to different subsequent addresses. Too high, and we burden the honeypots with a great deal of redundant traffic.

In principle, knowledge of N also allows an adversary to design their probes to escape capture by the honeyfarm. However, to do so the adversary also needs to know the address space over which we apply the threshold of N, and if they know that, they can more directly escape capture.

As a way to offset both these considerations, GQ's architecture supports *sampling* of the probe stream: taking a small, randomly selected subset of the stream and accepting it for processing without applying any of our filtering criteria. (See "Filter Bypass" in Figure 1.) We also use this mechanism for sources deemed "interesting" during the filtering stages, such as those that deviate from the replay proxy's scripts (see §3.4).

Our current implementation of this prefilter maintains state for each IP address that contacts our address ranges. If we found that this consumed too much state, however, we would switch to the approach from our AC-TRW hardware design [28], which uses an associative cache with an "evict-least-suspicious" policy. To date, we have not seen significant memory issues with keeping complete state.

GQ uses a second prefilter, too, taken from the work of Bailey *et al* [1]. After acknowledging an initial SYN packet, we examine the first data packet sent by the putative attacking host. If this packet unambiguously identifies the attack as a known and classified threat, the filter drops the connection. While configuration of these filters requires manual assessment (since many attacks *cannot* be unambiguously identified by their first data packet), we can use this mechanism to weed out background radiation from endemic sources such as Code Red and Slammer.

Note that our use of TCP header rewriting when instantiating our honeypots greatly simplifies use of this second prefilter. Without it, we would have to allocate a honeypot upon the initial SYN (if it passes the scan-N prefilter) so it could complete the SYN handshake, agree upon sequence numbers, and elicit the first data packet, just in case that packet proved of interest and we wished to continue the connection. We could not proxy for the beginning of the connection because any honeypot we might wind up instantiating later to handle the remainder of the connection would not have knowledge of the correct sequence numbers.

Additionally, we include a final "safety" filter, used only for filtering outbound traffic. Its role is to limit external traffic from the honeyfarm if any other component fails. This is a more relaxed but simpler policy than we implement in the rest of the honeyfarm, as the point is to assure—via independent operation—that any failure of our more complicated containment mechanism will not have a deleterious external effect.

3.4 Replay Proxy

Something quite important to realize (which we quantify in §5) is that the simple prefilters discussed in the previous section lack the power to identify many instances of known attacks. As an extreme example, the *W32.Femot* worm goes through *90 pairs* of message exchanges before we can distinguish one variant from another, because it is only at that point that the attack reveals the precise code it injects into the victim. Even for well-known malcode such as Blaster, the first-packet filter will fail to recognize new and potentially interesting variants, because these will not manifest until later. Yet creating a custom filter for each significant piece of known malcode is quite impractical due to their prevalence.

To filter these frequent multi-stage attacks, we employ technology developed for RolePlayer, an application-independent replay system [4]. RolePlayer uses a *script* automatically derived from one or two examples of a previous application session as the basis for mimicking either the client or the server side of the session in a subsequent instance of the same transaction. To do so, RolePlayer uses a series of heuristics to detect fields in application data that require modification for replaying the dialog. These include system names, IP addresses, dynamic ports, length fields, and transaction "cookies". RolePlayer can update these fields in the context of participating in a new session (without having any knowledge of the specific application coded into it), as well as opening or accepting additional connections when called for by the

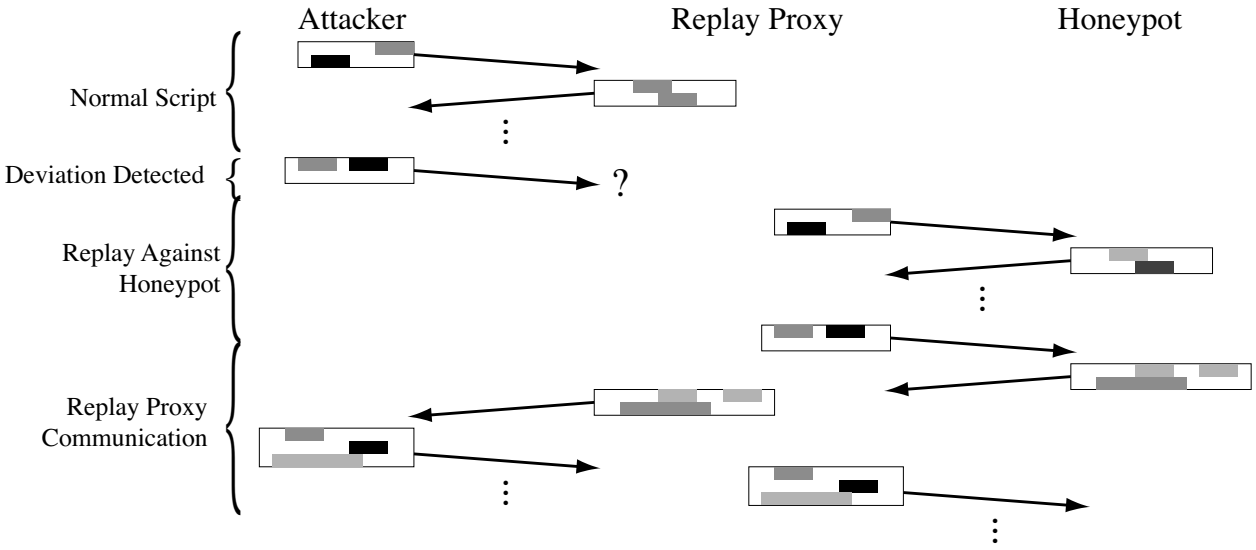


Figure 2: The Replay Proxy's operation. It begins by matching the attacker's messages against a script. When the attacker deviates from the script, the proxy turns around and uses the client side of the dialog so far to bring a honeypot server up to speed, before proxying (and possibly modifying) the communication between the honeypot and attacker. The shaded rectangles show in abstract terms portions of messages that the proxy identifies as requiring either echoing in subsequent traffic (e.g., a transaction identifier) or per-connection customization (e.g., an embedded IP address).

examples—vital for replaying multi-stage attacks.

The idea is that we can represent each previously seen attack using a script that describes the network-level dialog (both client and server messages) corresponding to the attack. A key property of such scripts is that RolePlayer can automatically extract them from samples of two dialogs corresponding to a given attack, *without* having to code into RolePlayer any knowledge of the semantics of the application protocol used in the attack. We leverage this feature of RolePlayer to build up a set of scripts with which we configure our *replay proxy*.

The replay proxy extends RolePlayer's functionality in three substantial ways: processing multiple scripts concurrently; recognizing when a source deviates from this set of scripts; and serving as an intermediary between a source and a high-fidelity honeypot once the script no longer applies.

The proxy first plays the role of the server side of the dialog, replying to incoming traffic from a remote source. By matching the source's messages against multiple scripts, the proxy can perform a fine-grained classification of known attacks without relying on application-specific responders. If the source never deviates from the set of scripts, then we have previously seen its activity, and we can drop the connection.

If the dialog deviates from the scripts, however, the proxy has found something that is in some fashion new. At this point it needs to facilitate cutting the remote source over from the faux dialog with which it has engaged so far to a continuation of that dialog with a high-fidelity honeypot. Doing so requires bringing the honeypot "up to speed." That is, as far as the source is concerned, it is already deeply engaged with a server. We need to synch the server up to this same point in the dialog so that the rest of the interaction can continue with full fidelity.

To do so, the proxy replays the dialog as it has received it so far back to the honeypot server; this time, the proxy plays the role of the client rather than the server's role. Once it has brought the honeypot up to speed, the source can then continue interacting with the honeypot. However, the replay proxy needs to remain engaged in the dialog as an intermediary, translating the IP addresses, sequence numbers, and, most

importantly, dynamic application fields (e.g., IP addresses embedded in application messages, or opaque "cookie" fields). Figure 2 shows the progression of the attacker ↔ replay-proxy ↔ honeypot-server dialog.

For new attacks, there exists a risk that the replay proxy might interfere with their execution because its knowledge is limited to scripts generated from previous attacks. Since the proxy has no semantic understanding of the communication relayed between the attacker and the honeypot, it cannot recognize if such interference occurs. Thus, the proxy marks the source of each attack it attempts to pass along to a backend honeypot as "interesting", such that any further connections from that source will be routed directly to a honeypot server, bypassing the replay proxy.

The replay proxy is one of GQ's most significant components. We discuss some implementation considerations for it in §4.1. Although we have not yet fully integrated the proxy into our operational GQ system, we have evaluated its effectiveness using both stand-alone tests and trace-based simulation (§5.2), finding that it offers great promise for offloading the honeypot servers.

3.5 Containment and Redirection

Another important design element concerns monitoring and enforcing the *containment policy*: i.e., as compromised honeypots initiate outbound network connections, do we allow these to traverse into the public Internet, redirect them to a fake respondent running inside the honeyfarm, or block them entirely? As noted above, the fundamental tension here is the need to balance between allowing contagion to exhibit its complete infection process (necessary for identifying self-propagation), but preventing the contagion from damaging external hosts.

To do so, we work through the following policy, in the given order:

1. If an outbound request would exceed the globally enforced, per-instantiated-honeypot rate limit → drop request. This acts as a general safety feature for containment, complementary to, but independent of, the "safety filter" previously discussed.

2. If an outbound DNS request originates from the DHCP/DNS-server honeypot → allow request. We configure GQ's honeypots with the address of a DHCP/DNS server that itself runs on a GQ honeypot. We allow that one node to make outbound DNS requests.

3. If outbound DNS request from regular honeypot → redirect request to DHCP/DNS honeypot. Additionally, we consider such requests as suspicious, since the honeypot's configuration should have originally sent them to the DHCP/DNS honeypot.

4. If the honeypot has not been probed → drop request. If a honeypot creates spontaneous activity (such as automatic update requests or similar tasks, beyond the DHCP we use for honeypot configuration), we simply block this activity. This serves both as a safety net and to ensure we know the exact configuration of the honeypot.

5. If outbound to the address of the source that caused GQ to instantiate the honeypot → allow connection. We allow the honeypot to always communicate with the source of the initial attack, to enable it to complete some forms of multi-stage infections and to participate in simple "phone home" confirmations.

6. If this is the first outbound connection to a host *other* than the infecting source → allow connection. Quite a bit of malcode uses a 2-stage infection process, where an initial infection then fetches the bulk of the contagion from a second remote site, or "phones home" to register the successful infection.[1]

[1]For example, we received notification that three of our telescope addresses appeared in a list produced upon the arrest of the suspected operators of the *toxbot* botnet [27]. Unlike many erroneous notifications we receive due to our use of the address space, this one was correct: our honeypots had indeed become infected by *toxbot*, and our containment policy correctly allowed them to contact the botnet's control infrastructure.

7. If the destination port is 21/tcp, 80/tcp, 443/tcp, or 69/udp AND the port was not the one accessed when the original remote source probed the honeypot AND the request is below a configurable threshold of how many such extra-port requests to allow → allow connection. Again, we do this to enable multi-stage infections and their control traffic, while attempting to limit the probability of an escaped infection.

8. If the request is a DHCP address-request broadcast → redirect it to DHCP/DNS honeypot.

9. Otherwise → redirect to a new honeypot if available, otherwise drop. Note that this step is critical for detecting self-propagating behavior, and highlights the importance of not running the honeyfarm at full capacity such that such connections fail to find honeypots available so they can manifest self-propagation.

In addition, when this process yields a decision to redirect, GQ consults a configurable *redirection limit* (computed on a per-instantiated-honeypot basis). If this connection would cause the honeypot to exceed the limit, we drop the connection instead, to prevent runaway honeypots from consuming excessive honeyfarm resources.

It is while assessing containment/redirection that GQ also makes a determination, from a network perspective, that we have spotted a new worm. We consider a system as infected with a worm if, when redirected to another honeypot, it causes *that* honeypot to begin initiating connections. Thus, from a network perspective, a worm is a program which, when running on one honeypot, can cause other honeypots to change state sufficiently that they also begin generating connection requests. This definition gives us a behavioral way to separate an infection that causes non-self-propagating network activity from one that self-propagates at least one additional step (thus distinguishing worms from botnet "autorooters" that compromise a machine and perhaps download configurable code to it, but do not cause it to automatically continue the process).

Redirection within the honeyfarm also allows us to perform malware classification. As the worm continues to generate requests, we redirect these to honeypots with differing configurations, including patch levels and OS variants. Doing so automatically creates a system *vulnerability signature*, determining which systems and configurations the worm can exploit.

The policy engine also manages the mapping of external (telescope) addresses to honeypots. Currently, when doing so it allocates different honeypots to distinct attackers, even if those attackers happen to target the same telescope address. Doing so keeps GQ's analysis of exactly what remote traffic infected a given honeypot, but also makes the honeyfarm vulnerable to fingerprinting by attackers.

A final facet of containment concerns internal containment. We need to ensure that infected honeypots cannot directly contact any other honeypot in the honeyfarm without our control system mediating the interaction. To do so, we require that all honeypot-to-honeypot communication pass through the control system, which also allows us to maintain each honeypot behind a NAT device.

We use VLANs as the primary technology for internal containment. This imposes a requirement that the back-end system running the honeypot VM can terminate a VLAN. We assign each honeypot to its own VLAN, with the GQ controller on a VLAN trunk. The controller can then rewrite packets on a per-VLAN as well as per-IP-address basis, enabling fine-grained isolation and control of all honeypot traffic, even broadcast and related traffic.

3.6 Honeypot Management

To maintain modularity, the GQ honeypot manager (per Figure 1) isolates management of honeypot configurations. The controller keeps an inventory and state of the different available honeypots, but defers direct operation of the VM honeypots to the honeypot manager, which is responsible for starting and reseting each honeypot. Only the honeypot manager needs to understand the VM or bare-metal technology used to implement a given honeypot. By doing this, we not only avoid the risk of inconsistency between the controller and the (stateless) honeypot manager, but also make the controller independent of the back-end implementation.

In the large, the GQ controller simply monitors the honeypots and responds to network requests. When the manager restarts a honeypot, the honeypot exists in a newly born state until requesting its IP address through DHCP. After the honeypot received an address assignment, the controller waits until it sees the system generate some network traffic (which it inevitably does as part of its startup). At this point the controller waits 10 seconds (to enable remaining services to start up and to avoid startup transients) before considering the honeypot available. Once available, the controller can now allocate the honeypot to serve incoming traffic. When later the controller decides that the instantiated honeypot no longer has value, it instructs the manager to destroy the instance and restart the honeypot in a reinitialized state.

4 Implementation Issues

In this section we describe the most significant issues that arose when realizing the GQ system as an implementation of the architecture presented in the previous section. These issues concerned the replay proxy system, the allocation and isolation mechanisms, and the VMware virtual machine management, which we detail in this section.

4.1 Replay Proxy

The replay proxy plays a central role in our implementation. Our current system takes a series of scripts for different attacks, with two sample dialogs (a "primary" dialog and a "secondary" dialog, in the terminology of [4], necessary for locating some types of dynamic fields) for each script. The main challenges for implementing the replay proxy are generating scripts for newly seen attacks and determining when a host has deviated from a script.

To generate a script for a newly seen worm attack, we take two different approaches depending on whether the attack succeeded or failed. For successful attacks, we can use the instances of infections redirected inside the honeyfarm for the primary and secondary dialogs. In this case, we also have control over replay-sensitive fields such as host names and IP addresses, which RolePlayer needs to know about to construct an accurate script.

For unsuccessful attacks, however, we have only one example of the application dialog corresponding to the attack. In this case, we face the difficult challenge of trying—without knowledge of the application's semantics—to find another instance of the exact same attack (as opposed to merely a variant or an attack that has some elements in common). We leverage a heuristic to tackle this problem: we assume that two very similar attacks from the same attacking host are in fact very likely the same attack, and we can therefore use the pair as the primary and secondary dialogs required by RolePlayer. We test whether two candidate attacks from the same host are indeed "very similar" by checking the number of Application Data Units (ADUs) in the two dialogs, their sizes, and the dynamic fields we locate within them. We require that the number of ADUs must be the same; the difference in ADU size quite small; and dynamic fields consistent across the two dialogs.

We use this technique to automatically construct a corpus of known types of activity. After a single infection by a worm, we want to automatically inform the replay proxy of the worm's attack to allow it to filter out subsequent probes from the same worm (which could become huge very rapidly, if the worm spreads quickly), in order to avoid tying up the honeyfarm with now-uninteresting additional worm traffic. We also benefit significantly from having scripts to filter out uninteresting instances of "background radiation."

To efficiently filter out known activity, rather than comparing an instance with all of the scripts at each step, we merge the scripts into a tree structure. For N scripts, doing so reduces the computational complexity from $O(N)$ to $O(\log(N))$. Constructing the tree requires comparing ADUs from different scripts. We decide that two ADUs from different scripts are the same if the match of dynamic fields between them is at least as good as that between the primary and secondary dialogs used to locate the fields in each script in the first place.

In online filtering, we use the same approach to check if a received ADU matches the ADU of a node in the tree. If not, it represents a *deviation* from our collection of scripts.

When we compare an ADU with a set of sibling nodes, it is possible that more than one of the nodes matches. When this happens, we choose the node that minimizes the number of *don't-care* fields (opaque data segments that appear in only one side of the attacks; see [4] for more details) identified in the compared node.

To avoid comparing a received ADU with every node in a large set of sibling nodes (which can arise for some common attack vectors), we use a Most Recently Used (MRU) technique to control the number of nodes compared. The intuition behind this technique is to leverage the locality of incoming probes. Given a set of sibling nodes, we associate with each node a time reflecting when it last matched a a received ADU. We compare newly received ADUs with those nodes having the most recent times first, to see if we can quickly find a close match. We find that in practice this works best if we compare with at least the first three nodes.

The replay proxy also supports detection of attack variants for attacks that involve multiple connections. For example, if a new Blaster worm uses the same buffer overflow code as the old one, but changes the contents of the executable file then transferred over TFTP, the replay proxy can correctly follow the dialog up through the TFTP connection, retrieve the executable file, notice it differs from the script at that point, and proceed to replay this full set of activity to a honeypot server in order to bring it up to speed to become infected by the attack variant.

A final detail is that when receiving data the proxy must determine when it has received a complete ADU. Since it may interact with unknown attacks, it cannot always directly tell when this occurs, so we use a timeout to ensure it reaches a decision. If the source has not sent any further packets when the timeout expires, we assume the data received until that point comprises an ADU.

Our replay proxy implementation comprises 7,500 lines of C code. It currently uses the socket APIs to communicate with the attackers and the honeypots. This limits its performance and makes it unwieldy to implement the address and sequence number translation it needs to perform. We will soon replace the socket APIs with a user-level transport layer stack (using raw sockets) and port the replay proxy to Click. Doing so will let us closely integrate the replay proxy with the other components of the honeyfarm controller, at which point we will run it as a production part of the system.

4.2 Isolation

Another critical component for our system implementation is maintaining rigorous isolation of all honeypots. The honeypots need to contact other honeyfarm systems—in particular, DHCP and DNS servers—but we need absolute control over these and any other interactions. Additionally, we need to keep our control traffic isolated from honeypot traffic.

To do so, we use VLANs and a VLAN-aware HP ProCurve 2800 series managed Ethernet switch [8]. We use a "base" VLAN with no encapsulation for a port on the controller and a port on each VMware ESX server. This LAN carries the management traffic for all the ESX servers, including access to our file server. By using an entirely separate (virtual) network with a switch capable of enforcing the separation, we can prevent even misbehaving honeypots from saturating our control plane.[2]

Equally important is maintaining isolation among honeypot traffic. Each individual honeypot should only communicate with authorized systems, which we accomplish using VLAN tagging and the Virtual Switch Tagging (VST) mode provided by the VMware ESX server. In this mode, each virtual switch in the ESX server will tag (untag) all outbound (inbound) Ethernet frames, leaving VLAN tagging transparent to the honeypots. This is important because we want to repeatedly use a single virtual machine image file for multiple running instances, and thus we want to avoid embedding any VLAN-related state in the image file. With this approach, the only untagged link is between the honeypot and the virtual switch; we achieve isolation for this segment by dedicating a port group (i.e., VLAN identifier) to each honeypot. Thus, the GQ controller's interface to the honeypot network receives VLAN-tagged packets, with each honeypot on a different VLAN.

[2]We are susceptible to bugs in the switch, however, if the VLAN encapsulation fails. We could instead use two separate switches if this proves problematic.

We also need to consider VLAN tagging/untagging for the controller. We implemented a VLAN manager in the controller, which maintains the mapping between IP addresses and VLAN IDs, and adds/removes VLAN tags before the Ethernet frames leave/enter the GQ controller. This approach allows us to completely abstract the isolation mechanism from the rest of the system.

Additionally, the VLAN manager has a "default route" for forwarding DHCP packets to the dedicated DHCP/DNS VM system. Rather than running these servers on the controller, where they might be susceptible to attack by a worm, we use a separate Linux "honeypot". By doing so we can both provide DHCP and DNS service for the honeypots, and detect attempts to attack DHCP or DNS servers once an initial honeypot system is compromised.

4.3 Honeypot Management

We have manually installed 10 different virtual machine images for both Windows and Linux (Fedora Core 4). The Windows images include Windows 2000 Professional; Windows 2000 Server; and Windows XP Professional with no patches, milestone-patches (e.g., Service Pack 2 for Windows XP), and fully patched. For ease of initial deployment, we operate only a subset of these which together cover a wide range of vulnerabilities in Windows systems.

We implemented the honeypot manager in C using the (undocumented other than by header files) vmcontrol interface provided by VMware. The communication between the honeyfarm controller and the honeypot manager follows a simple application-level protocol by which the controller instructs the manager what actions to take on what virtual machines, and the manager informs the controller of the results of those actions.

5 GQ Evaluation

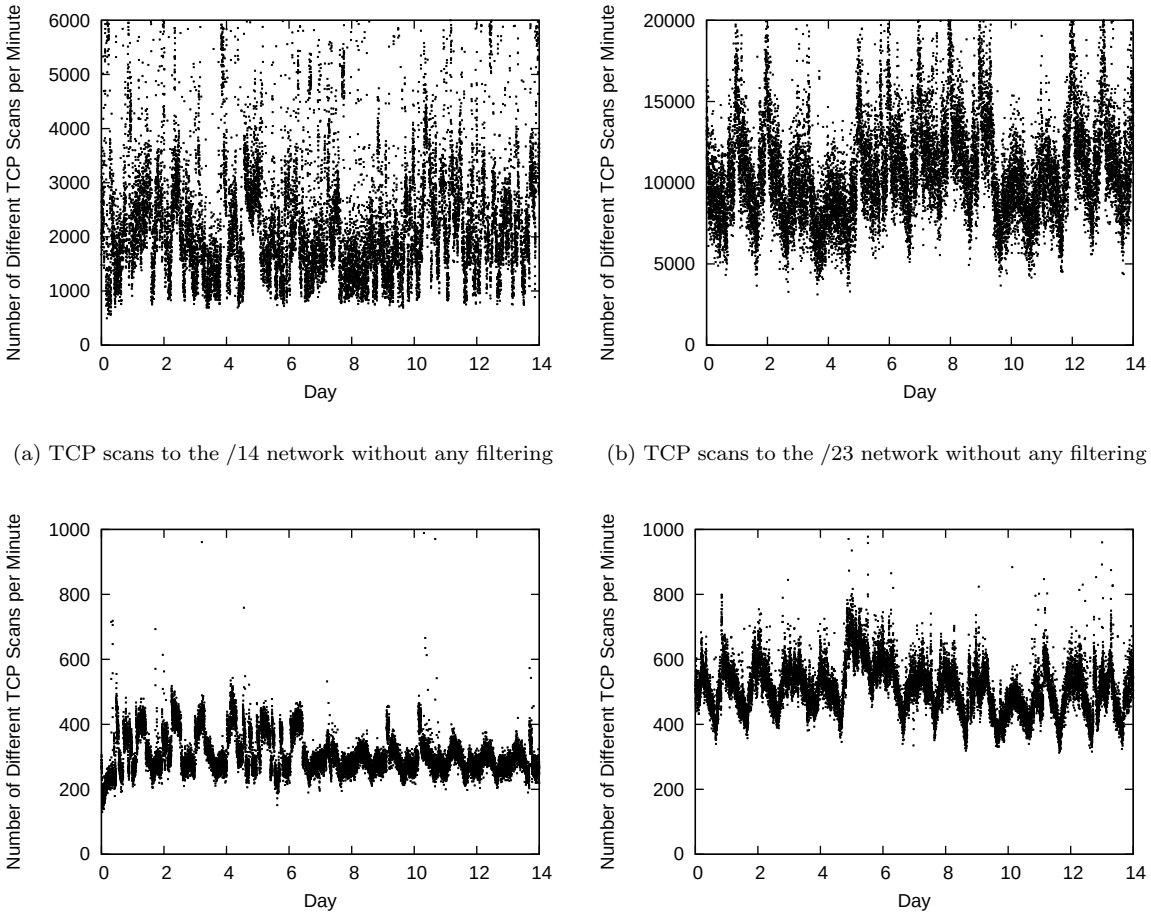
We began running GQ operationally in late 2005. The initial deployment we operate consists of four VMware ESX servers (soon to expand) running 24 honeypots in three different virtual machine configurations, including unpatched Windows XP Professional, unpatched Windows 2000 Server, and a fully patched Windows XP Professional system with an insecure configuration and weak passwords. Thus, GQ can currently capture worms that exploit Windows vulnerabilities on 80/tcp, 135/tcp, 139/tcp, and 445/tcp, but not other types of worms.

For our evaluation, we first examine the level of background radiation our network telescope captures and analyze the effectiveness of scan filtering for reducing this volume. We then evaluate the correctness and performance of the replay proxy. Finally, we present the worms GQ captured over the course of a month of operation.

5.1 Background Radiation and Scalability Analysis

Our network telescope currently captures the equivalent of a /14 network block (262,144 addresses), plus an additional "hot" /23 network (512 addresses). This latter block lies just above one of the private-address allocations. Any malware that performs sequential scanning using its local address as a starting point (e.g., Blaster [26]) and runs on a host assigned an address within the private-address block will very rapidly probe the adjacent public-address block, sending traffic to our telescope. Thus, this particular telescope feed provides *magnified* visibility into the activity of such malware.

We analyzed two weeks of network traffic recorded during honeyfarm operation. We find a huge bias in background radiation. Figures 3(a) and 3(b) show the number of TCP scans per minute seen by each telescope feed. Here, a "scan" means a TCP SYN from a distinct source-destination pair, ignoring any duplicate pairs that arrive within 60 seconds. We see that despite being 1/512th the size, the /23 network sees *several times more* background radiation than does the /14 network, dramatically illustrating the opportunity it provides to capture probes from sequentially scanning worms early during their propagation.



(a) TCP scans to the /14 network without any filtering

(b) TCP scans to the /23 network without any filtering

(c) TCP scans to the /14 network after scan filtering

(d) TCP scans to the /23 network after scan filtering

Figure 3: TCP scans before and after scan filtering

Figures 3(c) and 3(d) show the reduction in the raw probing achieved by prefiltering the traffic to remove scans. Our prefilter's current configuration allows each source to scan at most $N = 4$ different telescope addresses within an hour. We picked a cutoff of 4 because we currently run three different honeypot configurations, and the GQ controller allocates honeypots with different configurations to the same source when it probes additional telescope addresses; we then chose $N = 4$ rather than $N = 3$ to add some redundancy.

This filtering has the greatest effect on traffic from the /23 network, which by its nature sees numerous intensive scans, for which the prefilter discards all but the first few probes. As a result, after prefiltering the honeyfarm as a whole receives over a dozen attack probes/sec.

We used trace-based analysis to study the impact of different parameters for the scan filtering. Figure 4 shows for the /14 and /23 network the proportion of the incoming traffic that passes the prefiltering stage for a given setting of N, the filter cutoff. The cutoff specifies number of different telescope addresses allowed for a remote source to probe for a given period of time (as shown by the different curves). For the /23 network, the pass-through proportion remains around 5%. For the /14 network, it is considerably higher, but prefiltering still reaps significant benefits in reducing the initial load.

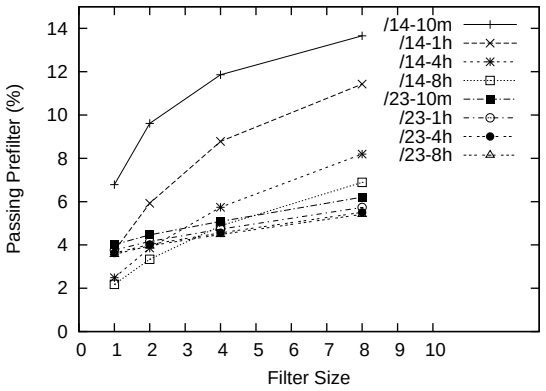


Figure 4: Effect of scan filter cutoff on proportion of probes that pass the prefilter.

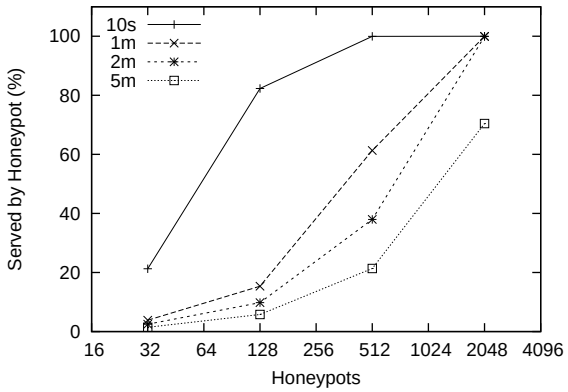


Figure 5: Effect of number of honeypots on proportion of prefiltered probes that can engage a honeypot.

However, even with aggressive prefiltering, more than 8 attack probes pass the filtering every second. This level rapidly consumes the available honeypots. In our current operation, we use 20 of the 24 honeypots to engage remote attackers, and the other 4 for internal redirection. Since engaging an attacker takes a considerable amount of time (including restarting the honeypot afresh after finishing), the honeyfarm is always saturated. To investigate the number of honeypots required in the absence of further filtering, we conducted a trace-based analysis using different mean honeypot "life cycles", where the life cycle consists of the attack engagement time plus the restart time. We varied the life cycle from 10 seconds up to 5 minutes (note that Xen-like virtual machines may take only seconds [25] to restart, while a full restart for a VMware virtual machine may take a full minute).

Figure 5 shows that once the life cycle exceeds 1 minute, we would need over 1,000 honeypots to engage 90% of the attacks that pass the prefiltering stage. Consequently, we require the second stage of filtering provided by our replay proxy, which we assess next.

5.2 Evaluating the Replay Proxy

In this section we assess the correctness and efficacy of the replay proxy as a means for further reducing the volume of probes that the honeypots must process.

To assess correctness, we extracted 717 attacks that successfully infected the honeypots and exhibited self-propagation inside the honeyfarm (detailed in §5.3). For each of these attacks, we validated the proxy's proper operation as follows:

1. We verified that the proxy can successfully replay the attack to compromise another vulnerable honeypot.

2. We ran the proxy with an incomplete script for the attack that only specifies part of the attack activity. We then launched the attack against the honeyfarm to confirm that the proxy would correctly follow the script through its end, and upon reaching the end would then successfully bring a vulnerable honeypot "up to speed" and mediate the continuation of the attack so that infection occurred.

3. We ran the proxy with a complete script of the attack to confirm that it successfully filtered further instances of the attack.

Of these, the first two are particularly critical, because incorrect operation can lead to either underestimating which other system configurations a given worm can infect (since we use replay to generate the test instances for constructing these vulnerability profiles), or failing to detect a novel attack (because the proxy does not

manage to successfully transfer the new activity to a honeypot). Incorrect operation in the third case results in greater honeyfarm load, but does not undermine correctness.

Port	Nodes	Max Depth	Max Breadth	Pairs of Examples
80/tcp	28	5	7	39
135/tcp	548	176	13	914
139/tcp	312	55	13	63
445/tcp	3,977	104	42	1,551

Table 1: Summary of protocol trees.

Because we have not yet integrated the replay proxy into the operational honeyfarm, to assess its filtering efficacy we constructed a set of scripts for it to follow and then ran those against attacks drawn from traces of GQ's operation. To do so, we first extracted 42,004 attack events for which GQ allocated a honeypot. (Note that each attack might involve multiple connections.) We then followed the approach described in §4.1 to automatically find 2,572 pairs of same attacks. Using these pairs as primary/secondary dialogs, we generated scripts for each and constructed four protocol trees for filtering, one for each of the four services accessed in the attack traces: 80/tcp, 135/tcp, 139/tcp, and 445/tcp. Table 1 shows the properties of the resulting trees.

Using this set of scripts to process the remaining 36,860 attack events, the proxy filters out 76% of them, indicating that operating it would offload the honeypots by a factor of 4, a major gain.

Finally, it is difficult to directly assess the false positive rate (attacks that should not have been filtered out but were), because doing so entails determining the equivalence of the actual operational semantics of different sets of server requests. However, we argue that the rate should be low because we take a conservative approach by requiring that filtered activity must match all dynamic fields in a script, including large ADUs that correspond to buffer overruns or bulk data transfers.

5.3 Capturing Worms

In four months of operation GQ automatically captured 66 distinct worms belonging to 14 different families. Appendix A summarizes the different types, where a worm type corresponds to a unique MD5 checksum for the worm's executable code. Most of these executables have a name directly associated with them because they are uploaded as files by the worm's initial exploit (see below). The table also gives the size in bytes of the executable and how many times GQ captured an instance of the worm.

To identify the worms, we uploaded the malicious executables to a Windows virtual machine on which we installed Symantec Antivirus. While its identification is incomplete (and some of the names appear less convincing), since we lack access to a large worm corpus we include the names for completeness.

We captured not only buffer-overflow worms but also those that exploit weak passwords. All of those worm attacks required multiple connections to complete (as many as 72 for BAT.Boohoo.Worm), as listed in the penultimate column, and involved a data transfer channel separate from the exploit connection to upload the malicious executable to the victim. This nature highlights the weakness of using "first payload" techniques for filtering out known attacks from background radiation: a large number of attacks only distinguish themselves as novel or known after first engaging in significant initial activity.

The table also shows the (minimum) detection time for each worm. We measure detection time as the interval between when the first scan packet arrived at the honeyfarm, to when a *second* honeypot (infected by redirection from the first honeypot that served the request itself) attempts to make an outbound connection attempt, which provides proof that we have captured code that self-propagates. Detection time depends on many factors: end host response delay, network latency, execution time of the malware within the honeypot, and redirection honeypot availability. (We inspected network traces to confirm that delay due to processing by the GQ controller is negligible.)

While the times given in Appendix A seem high, often the culprit is a slow first stage during which the remote source initially infects a honeypot. When we tested GQ's detection process by releasing Code Red and Blaster within it, the detection time was around one second.

6 Future Work

There are four major areas we are pursuing as near-term future work.

First, we aim to soon expand the system in terms of additional honeypots (for which we have new hardware now ready), breadth of system and server images to run on them, and network telescope reach.

Second, we need to integrate the replay proxy into GQ's operations. As shown in §5.2, we expect this to roughly quadruple the honeyfarm's processing capacity by allowing GQ to avoid spending resources on activities semantically equivalent to those it has already processed in the past. Currently, a single-threaded instance of the replay proxy can process about 100 concurrent probes per second. To use the proxy operationally, we must improve its performance so that it can process significantly more concurrent connections. We aim to solve this problem from three perspectives: (1) improve the replay algorithm to reduce the computational cost; (2) leverage multiple threads/processes/machines; (3) convert its network input/output to use raw sockets. Our general goal is for the overhead of the replay proxy processing for an incoming probe to be much less than that for launching a virtual machine honeypot.

Third, the filtering efficacy of the replay proxy depends critically on the richness of the set of scripts it has for describing previously seen activity. To this end, we aim to fully automate the process of updating these scripts every time the honeyfarm processes a new type of activity. Doing so will enable GQ to both dynamically adapt its filtering and to automatically chart the spread of attacks and their variants.

Finally, we want to incorporate two forms of alternate backend platforms: lightweight Xen-based virtual machines [25], and "bare metal" honeypots that execute directly on hardware rather than inside a virtual machine. For these latter, we plan to draw upon techniques pioneered by Emulab [7] for reimaging and managing the honeypots. We believe that GQ's modular architecture should accommodate these changes quite readily.

7 Conclusions

Recently, great interest has arisen in the construction of *honeyfarms*: large pools of honeypots that interact with probes received over the Internet to automatically determine the nature of the probing activity, especially whether it signals the onset of a global worm outbreak.

Building an effective honeyfarm raises many challenging technical issues. These include ensuring high-fidelity honeypot operation; efficiently discarding the incessant Internet "background radiation" that has only nuisance value when looking for new forms of activity; and devising and policing an effective "containment" policy to ensure that captured malware does not inflict external damage or skew internal analyses.

In this work we have presented *GQ*, a high-fidelity honeyfarm system designed to meet these challenges. GQ runs fully functional servers across a range of operating systems known to be prime targets for worms (especially various flavors of Microsoft Windows), confining each server to a virtual machine to maintain full control over its operation once compromised. To cope with load, GQ leverages aggressive filtering, including a technique based on application-independent "replay" that holds promise to quadruple GQ's effective capacity.

Among honeyfarm efforts, our experiences with GQ are singular in that the scale at which we have operated it—monitoring more than a quarter million Internet addresses, and capturing 66 distinct worms in four months of operation—far exceeds that previously achieved for a high-fidelity honeyfarm. While much remains for pushing the system further in terms of capacity and automated, efficient operation, its future as a first-rate tool for analyzing Internet-scale epidemics appears definite.

References

[1] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor - a distributed blackhole monitoring system. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2005.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[3] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, Farnam Jahanian, and Danny McPherson. Toward understanding distributed blackhole placement. In *Proceedings of ACM CCS WORM*, October 2004.

[4] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[5] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. HoneyStat: Local worm detection using honeypots. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[6] Jeff Dike. The user-mode linux. http://user-mode-linux.sourceforge.net/.

[7] Emulab. http://www.emulab.net/.

[8] Hewlett-Packard. http://www.hp.com/rnd/products/switches/2800_series/overview.htm.

[9] Honeynet. The honeynet project. http://www.honeynet.org/.

[10] Xuxian Jiang and Dongyan Xu. Collapsar: A vm-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[11] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[12] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an Internet-scale event. In *Proceedings of the 2005 ACM Internet Measurement Conference*, October 2005.

[13] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[14] William Metcalf. Snort inline. http://snort-inline.sourceforge.net/.

[15] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Magazine of Security and Privacy*, August 2003.

[16] David Moore and Colleen Shannon. The spread of the witty worm. *IEEE Security and Privacy*, 2(4), July/August 2004.

[17] David Moore, Colleen Shannon, and Jeffery Brown. Code-red: a case study on the spread and victims of an Internet worm. In *Internet Measurement Workshop*, November 2002.

[18] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Network telescopes: Technical report. Technical Report Technical Report, CAIDA, April 2004.

[19] David Moore, Geoffrey Voelker, and Stefan Savage. Inferring Internet denial-of-service activity. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[20] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet background radiation. In *Proceedings of Internet Measurement Conference*, October 2004.

[21] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[22] Lance Spitzner. Honeypot farms. http://www.securityfocus.com/infocus/1720.

[23] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003.

[24] VMware Inc. http://www.vmware.com/.

[25] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[26] W32.Blaster.Worm. http://securityresponse.symantec.com /avcenter/venc/data/w32.blaster.worm.html.

[27] W32.Toxbot. http://securityresponse.symantec.com/avcenter/venc/data/w32.toxbot.html.

[28] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th Usenix Security Symposium*, San Diego, CA, August 2004.

[29] Vinod Yegneswaran, Paul Barford, and Vern Paxson. Using honeynets for Internet situational awareness. In *Proceedings of ACM SIGCOMM HotNets-IV Workshop*, November 2005.

[30] Vinod Yegneswaran, Paul Barford, and Dave Plonka. On the design and use of Internet sinks for network abuse monitoring. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

A Worms Captured

Executable Name	Size (B)	MD5Sum	Worm Name	# Events	# Conns	Time (s)
a####.exe	10366	7a67f7c8...	W32.Zotob.E	4	3	29.0
a####.exe	10878	bf47cfe2...	W32.Zotob.H	9	3	25.2
a####.exe	25726	62697686...	Quarantined but no name	1	3	223.2
cpufanctrl.exe	191150	1737ec9a...	Backdoor.Sdbot	1	4	111.2
chkdisk32.exe	73728	27764a5d...	Quarantined but no name	1	4	134.7
dllhost.exe	10240	53bfe15e...	W32.Welchia.Worm	297	4 or 6	24.5
enbiei.exe	11808	d1ee9d2e...	W32.Blaster.F.Worm	1	3	28.9
msblast.exe	6176	5ae700c1...	W32.Balster.Worm	1	3	43.8
lsd	18432	17028f1e...	W32.Poxdar	11	8	32.4
NeroFil.EXE	78480	5ca9a953...	W32.Spybot.Worm	1	5	237.5
sysmsn.exe	93184	5f6c8c40...	W32.Spybot.Worm	3	3	79.6
MsUpdaters.exe	107008	aa0ee4b0...	W32.Spybot.Worm	1	5	57.0
RealPlayer.exe	120320	4995eb34...	W32.Spybot.Worm	2	5	95.4
WinTemp.exe	209920	9e74a7b4...	W32.Spybot.Worm	1	5	178.4
wins.exe	214528	7a9aee7b...	W32.Spybot.Worm	1	5	118.2
msnet.exe	238592	6355d4d5...	W32.Spybot.Worm	1	7	189.4
MSGUPDATES.EXE	241152	65b401eb...	W32.Spybot.Worm	2	5	125.3
ntsf.exe	211968	5ac5998e...	Qurantined but no name	1	5	459.4
scardsvr32.exe	33169	1a570b48...	W32.Femot.Worm	4	3	46.2
scardsvr32.exe	34304	b10069a8...	W32.Femot.Worm	1	3	66.5
scardsvr32.exe	34816	ba599948...	W32.Femot.Worm	55	3	96.6
scardsvr32.exe	35328	617b4056...	W32.Femot.Worm	2	3	179.6
scardsvr32.exe	36864	0372809c...	W32.Femot.Worm	1	5	49.3
scardsvr32.exe	39689	470de280...	W32.Femot.Worm	4	3	41.4
scardsvr32.exe	40504	23055595...	W32.Femot.Worm	1	3	41.1
scardsvr32.exe	43008	ff20f56b...	W32.Valla.2048	1	5	32.2
scardsvr32.exe	66374	f7a00ef5...	Quarantined but no name	1	7	54.8
scardsvr32.exe	205562	87f9e3d9...	W32.Pinfi	1	3	180.8
x.exe	9343	986b5970...	W32.Korgo.Q	17	2	6.6
x.exe	9344	d6df3972...	W32.Korgo.T	7	2	9.5
x.exe	9353	7d99b0e9...	W32.Korgo.V	102	2	6.0
x.exe	9359	a0139d7a...	W32.Korgo.W	31	2	5.9
x.exe	9728	c05385e6...	W32.Korgo.Z	20	2	6.6
x.exe	11391	7f60162c...	W32.Korgo.S	169	2	6.6
x.exe	11776	c0610a0d...	W32.Korgo.S	15	2	8.6
x.exe	13825	0b80b637...	W32.Korgo.V	2	2	24.4
x.exe	20992	31385818...	W32.Licum	2	2	7.9
x.exe	23040	e0989c83...	W32.Korgo.S	3	2	10.4
x.exe	187348	384c6289...	W32.Pinfi	1	2	329.7
x.exe	187350	a4410431...	W32.Korgo.V	6	2	11.3
x.exe	187352	b3673398...	W32.Pinfi	5	2	20.1
x.exe	187354	c132582a...	W32.Pinfi	5	2	24.9
x.exe	187356	d586e6c2...	W32.Pinfi	2	2	27.5
x.exe	187358	2430c64c...	W32.Korgo.V	1	2	27.5
x.exe	187360	eb1d07c1...	W32.Pinfi	1	2	63.1
x.exe	187392	2d9951ca...	W32.Korgo.W	1	2	76.1
x.exe	189400	7d195c0a...	W32.Korgo.S	1	2	18.0
x.exe	189402	c03b5262...	W32.Pinfi	1	2	58.2
x.exe	189406	4957f2e3...	W32.Korgo.S	1	2	210.9
xxxx...x	46592	a12cab51...	Backdoor.Berbew.N	844	2	9.4
xxxx...x	56832	b783511e...	W32.Info.A	34	2	7.2
xxxx...x	57856	ab5e47bf...	Trojan.Dropper	685	3	10.0
xxxx...x	224218	d009d6e5...	W32.Pinfi	1	3	32.5
xxxx...x	224220	af79e0c6...	W32.Pinfi	3	2	34.2
n/a	10240	7623c942...	W32.Korgo.C	3	2	4.8
n/a	10752	1b90cc9f...	W32.Korgo.L	1	2	7.0
n/a	10752	32a0d7d0...	W32.Korgo.G	8	2	4.1
n/a	10752	ab7ecc7a...	W32.Korgo.N	2	2	5.3
n/a	10752	d175bad0...	W32.Korgo.G	3	2	5.4
n/a	10752	d85bf0c5...	W32.Korgo.E	1	2	5.6
n/a	10752	b1e7d9ba...	W32.Korgo.gen	1	2	5.0
n/a	10879	042774a2...	W32.Korgo.I	15	2	4.3
n/a	11264	a36ba4a2...	W32.Korgo.I	1	2	5.4
multiple	n/a	n/a	W32.Muma.A	2	7	186.7
multiple	n/a	n/a	W32.Muma.B	2	7	208.9
multiple	n/a	n/a	BAT.Boohoo.Worm	1	72	384.9

Summary of captured worms (worm names are reported by Symantec Antivirus).