

GQ: Practical Containment for Measuring Modern Malware Systems

Christian Kreibich,[†] Nicholas Weaver,[†] Chris Kanich,^{*}
Weidong Cui,[§] and Vern Paxson^{†‡}

TR-11-002

May 2011

Abstract

Measurement and analysis of modern malware systems such as botnets relies crucially on *execution* of specimens in a setting that enables them to communicate with other systems across the Internet. Ethical, legal, and technical constraints, however, demand *containment* of resulting network activity in order to prevent the malware from harming others while still ensuring that it exhibits its inherent behavior. Current best practices in this space are sorely lacking: measurement researchers often treat containment superficially, sometimes ignoring it altogether. In this paper we present GQ, a malware execution “farm” that uses explicit containment primitives to enable analysts to develop containment policies naturally, iteratively, and safely. We discuss GQ’s architecture and implementation, our methodology for developing containment policies, and our experiences gathered from six years of development and operation of the system.

[†] ICSI, 1947 Center St., Ste. 600, Berkeley, CA 94704

^{*} UC San Diego, San Diego, CA

[§] Microsoft Research, Redmond, Washington

[‡] UC Berkeley, Berkeley, CA 94704

1 Introduction

Despite strenuous efforts of security vendors and researchers, the Internet is not becoming a safer place. Recent industry reports state that 60,000 new samples of malware—software created with the intent to conduct malicious activity—appear every day [19]. In this environment, malware analysts face an increasingly challenging task of separating chaff—uninteresting rehashes of known malware strains—from truly novel behavior, in order to enable close study of the novelty to understand its goals and determine how to combat it. At the same time, analysts must also follow the activity in known, major outbreaks—akin to infiltrating criminal organizations—to stay abreast of recent developments. These tasks rely crucially on *execution*: “letting loose” malware samples in an execution environment to study their behavior, sometimes only for seconds at a time (e.g., to understand the bootstrapping behavior of a binary, perhaps in tandem with static analysis), but potentially also for weeks on end (e.g., to conduct long-term botnet monitoring via “infiltration” [13]).

This need to execute malware samples in a laboratory setting exposes a dilemma. On the one hand, unconstrained execution of the malware under study will likely enable it to operate fully as intended, including embarking on a large array of possible malicious activities, such as pumping out spam via email or the web, contributing to denial-of-service floods, conducting click fraud, or obscuring other attacks by proxying malicious traffic. On the other hand, if executed in complete isolation, the malware will almost surely fail to operate as intended, since it cannot contact external servers via its command-and-control (C&C) channel in order to obtain input data or execution instructions.

Thus, industrial-strength malware analysis requires *containment*: execution environments that on the one hand allow malware to engage with the external world to the degree required to *manifest* their core behavior, but doing so in a highly controlled fashion to *prevent* the malware from inflicting harm on others. Despite the critical importance of proper containment, researchers often treat the matter at best superficially, sometimes ignoring it altogether. We perceive the root cause for this shortcoming as arising largely from a lack of technical tools to realize sufficiently rich containment, along with a perception of containment as a chore rather than an opportunity. To improve the state of affairs, we present *GQ*, a malware execution “farm” we have developed and operated regularly for six years. *GQ*’s design explicitly focuses on enabling the development of precise-yet-flexible containment policies for a wide range of malware. Drawing upon *GQ*’s development and operation over this period, we make three contributions:

First, we develop an architecture for a malware execution platform that provides a natural way to configure and enforce containment policies by means of *containment servers*, establishing first-order primitives for effecting containment on a per-flow granularity. We began our work on *GQ* during the peak of the “worm era” [9], and initially focused on capturing self-propagating network worms. Accordingly, we highlight both the system’s original architectural features that have “stood the test of time,” as well as the shortcomings that emerged as our usage of *GQ* began to shift towards broader forms of malware.

Second, we describe a principled—albeit largely manual—approach to the development of containment policies to ensure prevention of harm to others. Our goal here is to draw attention to the issue, highlight potential avenues for future research, and point out that far from being a chore, containment development can actively *support* the process of studying malware behavior.

Third, we present insights from our extensive operational experience [4, 9, 13, 17, 18, 24] in developing containment policies that serve to illustrate the effectiveness, importance, and utility of a principled approach to containment.

We begin by reviewing the approaches to containment framed in previous work (§ 2), which provides further context for our discussion of the issue of containment (§ 3). We then present *GQ*’s design goals (§ 4), architecture (§ 5), and implementation (§ 6), illustrating both initial as well as eventual features as a consequence of our continued desire to separate policy from mechanism and increase general applicability of the system. Finally, we report on our operational experiences over *GQ*’s lifetime (§ 7), discuss our current methodology for developing containment policies and containment’s general feasibility (§ 8), and offer concluding thoughts (§ 9).

2 Related Work

A large body of prior work focuses on malware execution for the purpose of understanding the *modi operandi* of binary specimens. Historically, “honeyfarms” were the first platforms supporting large-scale malware execution. These systems focused in particular on *worm* execution, with the “honey” facet of the name referring to a honeypot-style external appearance of presenting a set of infectible systems in order to trap specimens of global worm outbreaks early in the worm’s propagation. Vrabie et. al designed the Potemkin honeyfarm as a (highly) purpose-specific prototype of a worm honeyfarm that explored the scalability constraints present when simulating hundreds of victim machines on a single physical machine [28]. Their work also framed the core issue of containment, but leveraged a major simplification that (sim-

ple) worms can potentially provide, namely that one can observe worm propagation even when employing a very conservative containment policy of redirecting outbound connections to additional analysis machines in the honeyfarm. Jian and Xu’s Collapsar, a virtualization-based honeyfarm, targeted the transparent integration of honeypots into a distributed set of production networks [11]. The authors focused on the functional aspects of the system, and while Collapsar’s design supports the notion of assurance modules to implement containment policies, in practice these simply relied on throttling and use of “Inline Snort” to block known attacks.

With the end of the “worm era,” researchers shifted focus from worm-like characteristics to understanding malware activity more broadly. Bayer et al. recognized the significance of the containment problem in a paper introducing the Anubis platform [2], which has served as the basis for numerous follow-up studies: *“Of course, running malware directly on the analyst’s computer, which is probably connected to the Internet, could be disastrous as the malicious code could easily escape and infect other machines.”* The remainder of the paper, however, focuses on technical aspects of monitoring the specimen’s interaction with the OS; they do not address the safety aspect further.

The CWSandbox environment shares the dynamic analysis goals of Anubis, using virtualization or potentially “raw iron” (non-virtualized) execution instead of full-system emulation [30]. The authors state that CWSandbox’s reports include *“the network connections it opened and the information it sent”*, and simply acknowledge that *“using the CWSandbox might cause some harm to other machines connected to the network.”*

The Norman SandBox [22] similarly relies on malware execution in an instrumented environment to understand behavior. As a proprietary system, we cannot readily infer its containment model, but according to its description it emulates the *“network and even the Internet.”*

Chen et al. [7] presented a “universe” management abstraction for honeyfarms based on Potemkin, allowing multiple infections to spread independently in a farm. Their universes represent sets of causally related honeypots created due to mutual communication. GQ provides such functionality in a more general fashion via its support for independently operating subfarms. In contrast to Potemkin’s universes, GQ’s subfarms do not imply that malware instances executing within the same subfarm necessarily can communicate.

More recently, John et al. presented Botlab [12], an analysis platform for studying spamming botnets. Botlab shares commonalities with GQ, including long-term execution of bot binaries, in their case particularly for the purpose of collecting spam campaign intelligence.

They also recognize the importance of executing malware safely. In contrast to GQ, the authors designed Botlab specifically to study email spam, for which they employed a static containment policy: *“[T]raffic destined to privileged ports, or ports associated with known vulnerabilities, is automatically dropped, and limits are enforced on connections rates, data transmission, and the total window of time in which we allow a binary to execute.”* In addition, the authors ultimately concluded that containment poses an intractable problem, and ceased pursuing their study: *“Moreover, even transmitting a “benign” C&C message could cause other, non-Botlab bot nodes to transmit harmful traffic. Given these concerns, we have disabled the crawling and network fingerprinting aspects of Botlab, and therefore are no longer analyzing or incorporating new binaries.”*

Researchers have also explored malware execution in the presence of complete containment (no external connectivity). These efforts rely on mechanisms that aim to mimic the fidelity of communication between bots and external machines. The SLINGbot system emulates bot behavior rather than allowing the traffic of live malware on the commodity Internet [10]; clearly, such an approach can impose significant limits on the obtainable depth of analysis. The DETER testbed [21] relies upon experimenters and testbed operators to come to a consensus on the specific containment policy for a given experiment. Barford and Blodgett’s Botnet Mesocosms run real bot binaries, but emulate the C&C infrastructure in order to evaluate bot behavior without Internet connectivity [1]. Most recently, Clavet et al. described their in-the-lab botnet experimentation testbed with a similar containment policy to Botnet Mesocosms, but with the capability of running thousands of bot binaries at one time [5]. While all of these systems guarantee that malicious traffic does not interact with the commodity Internet, they required substantial effort for enabling a useful level of operational fidelity and, in the process, necessarily sacrifice the fidelity that precisely controlled containment can offer.

3 The Problem of Containment

The frequently light (at best) treatment of the central problem of executing malware *safely* is to some extent understandable. Established tool-chains for developing containment do not exist, so the resulting containment mechanisms are ad-hoc or suboptimal, leaking undesirable activity or preventing desired activity. This deficiency renders containment implementation and verification a difficult, time-consuming problem—one that frequently gets in the way of conducting the intended experiment. GQ aims to fill this gap by making containment policy development a natural component of the malware

execution process.

Development of precise containment requires a balance of (i) *allowing* the outside interactions necessary to provide the required realism, (ii) *containing* any malicious activity, all while (iii) convincingly *pretending* to the sample under study that it enjoys unhampered interaction with external systems. Lacking rigor in this process raises ethical issues, since knowingly permitting such activity can cost third parties significant resources in terms of cleaning up infections, filtering spam, fending off attacks, or even financial loss. It also may expose the analyst to potential legal consequences, or at the very least to abuse complaints. Finally, technical considerations matter as well: prolonged uncontained malicious activity can lead to blacklisting of the analysis machines by the existing malware-fighting infrastructure. Such countermeasures can prevent the analyst from acquiring unbiased measurements as they may filter key activity elsewhere in the network, or because the malware itself changes its behavior once it detects suspiciously diminishing connectivity. We might consider full containment as a viable option for experiments such as “dry-runs” of botnet takedowns [5], but we find that frequently the dynamically changing nature of real-world interactions and the actual content of delivered instructions holds central interest for analysis.

Thus, we argue for the importance of avoiding the perception of containment as a resource-draining chore. Indeed, we find that developing containment from a default-deny stance, iteratively permitting understood activity, in fact provides an excellent way to understand the behavioral envelope of a sample under study. We discuss this further in § 8.

Finally, we emphasize that we do not mean for GQ to replace the use of static or dynamic malware analysis tools. Rather, we argue that researchers should deploy cutting-edge malware analysis tools *inside* a system such as GQ, to provide a mature containment environment in which they can employ sophisticated malware instrumentation safely and robustly.

4 Initial & Eventual Design Goals

As we mentioned above, our initial design for GQ targeted a classic, worm-focused honeyfarm architecture. Over the past years, we focused on evolving the system into a generic *malware farm*—a platform suitable for hosting all manner of malware-driven research safely, routinely, and at scale. In the following we outline original design goals as well as those we came to embrace at later stages of building the system.

Versatility. In contrast to previous work, GQ must serve as a platform for a broad range of malware experimentation, with no built-in assumptions about inmates

belonging to certain classes of malware, or acting exclusively as servers (realizing traditional honeyfarms) or clients (realizing honeycrawlers [29]). Our initial focus on worms imposed design constraints that became impractical once we wanted to experiment with non-self-propagating malware. Similarly, focus on a particular class of botnets (say those using IRC as C&C, or domain generation algorithms for locating C&C servers via DNS), restricts versatility.

Separation of policy and mechanism. This goal closely relates to the former, but we needed to invest considerable thought in order to achieve it in a satisfying fashion. In our original honeyfarm architecture, we tightly interwove mechanism (a highly customized packet forwarder) and policy (worm containment) in forwarding code that we could adapt only with difficulty and by imposing system downtime. GQ must clearly separate a comparatively stable containment enforcement mechanism from an adaptable containment policy definition.

Verifiable containment. GQ must provide complete control over all inbound and outbound communication. This remains as true now as it was at the outset of GQ’s development. Depending on a flow’s source, destination, and content the system must allow dropping, reflecting, redirecting, rewriting, and throttling the flow flexibly depending on current execution context. Moreover, mechanisms should exist to verify that developed policies operate as intended.

Multiple execution platforms. Malware frequently checks the nature of its execution platform in order to determine whether it likely finds itself running in an analysis environment or as a successful infection in the wild. For example, malware often considers evidence of virtualization a tell-tale sign for operation in a malware analysis environment [8] and ceases normal operation. GQ must support virtualized, emulated, and raw iron execution as desired, on a per-inmate granularity, in a fashion transparent to the containment process.

Multiple experiments. While extending GQ we increasingly noticed the need for running multiple malware-driven setups at the same time. The original *single-experiment* design made it difficult to accommodate this. For GQ, we aim to provide a platform for independently operating experiments involving different sets of machines, containment policies, and operational importance. For example, we have found it exceedingly useful to run a “deployment” and “development” setup for running spambots; one for continuously harvesting spam from specimen for which we have developed mature containment policies, the other one for developing such policies on freshly obtained samples.

Threat model. We need to consider the way in which machines on our inmate network become infected, as well as the behavioral envelope once a machine has be-

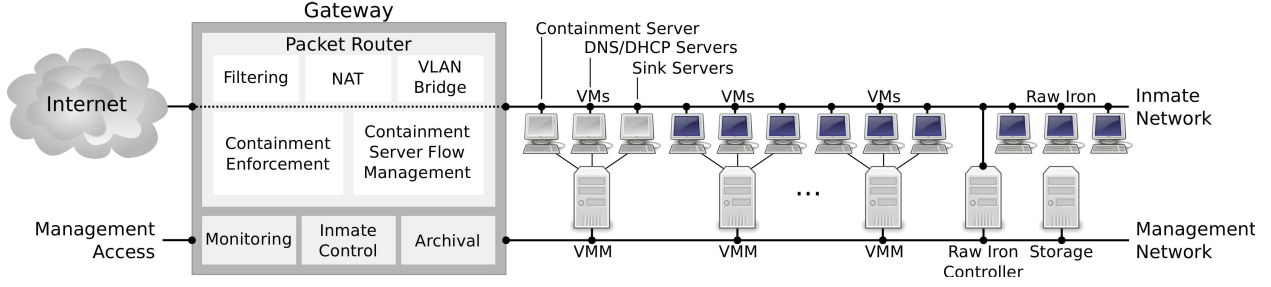


Figure 1: GQ’s overall architecture.

come infected. Regarding the former, if an experiment so desires, GQ fully supports the classic honeyfarm model in which external traffic directly infects honeypot machines. Today, we find *intentional infection* with malware samples collected from external sources more relevant and useful. We describe our mechanism for facilitating such infections in § 6.6. Regarding post-infection behavior, we adopt the position of other systems executing malware in virtualized environments and assume that the program cannot break out of the virtual environment. The malware may corrupt user and kernel space on the infected host arbitrarily and conduct arbitrary network I/O.

5 GQ’s Architecture

5.1 Overall layout

We show GQ’s conceptually simple architecture in Figure 1. A central *gateway* sits between the outside network and the internal machinery, separating it into an *inmate network* on which we host infected machines—*inmates*—and a *management network* that we use to access virtual machine monitors (VMMs) and other farm control infrastructure.

Within the gateway, a custom packet forwarding logic enables and controls traffic flow between the farm and the outside network, as well as among machines within the farm. This takes several forms. First, a learning VLAN bridge incrementally understands and enables crosstalk among machines on the inmate network as required subject to the containment policy in effect. Second, we devote the majority of the forwarding logic to the redirection of outgoing and incoming flows to a *containment server*, which decides what containment policies to enforce on a given flow, and to enforce these containment policies. Finally, a safety filter ensures that the rate of connections across destinations and to a given destination never exceeds a configurable threshold.

5.2 Inmate hosting & isolation

We employ three different inmate-hosting technologies: full-system virtualization (using VMware ESX servers), unvirtualized “raw iron” execution (more on this in § 6.4), and QEMU for customized emulation [4]. The hosting technology we employ for a given inmate remains transparent to the gateway.

GQ enforces inmate isolation at the link layer: each inmate sends and receives traffic on a VLAN ID unique on the inmate network. VLAN IDs thus serve as handy identifiers for individual inmates. Physical and, where feasible, virtual switches (for simplicity not shown in Figure 1) behind the gateway enforce the per-inmate VLAN assignment, which our inmate creation/deletion procedure automatically picks and releases from the available VLAN ID pool.

The gateway’s packet forwarding logic enables connectivity to the outside Internet as permissible, while selectively enabling inmate crosstalk as required.

5.3 Network layout

We use network address translation for all inmates. The inmate network uses non-routable RFC 1918 addresses, which the gateway maps to and from a configurable global address space. This provides one level of indirection and allows us to keep the inmate configuration both simple and dynamic at inmate boot time. To facilitate this, the inmate network provides a small number of *infrastructure services* for the inmates, including DHCP and recursive DNS resolvers. The gateway places these services into a restricted broadcast domain, comprising all machines the inmates can reach by default. In addition, infrastructure services include experiment-specific requirements such as SMTP sinks, though these do not belong to the broadcast domain.

5.4 Containment servers

GQ’s design hinges crucially on the notion of an explicit *containment server* that determines each flow’s contain-

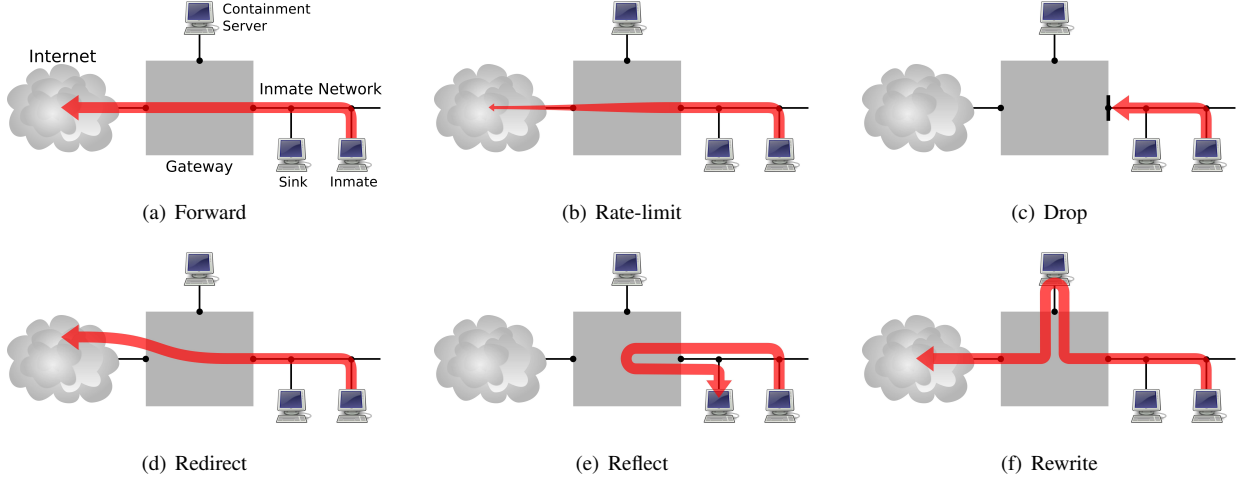


Figure 2: GQ’s flow manipulation modes, illustrated on flows initiated by an inmate.

ment policy and one of the major improvements over existing systems. Our original design included no equivalent; instead, the packet router in the gateway alone implemented the containment policy. Its design results from our desire to reduce the gateway’s forwarding logic to an infrequently changing *mechanism* to the greatest extent possible, while keeping containment *policies* configurable and adaptable. This contrasts starkly with our initial design, which intermixed a fixed nine-step containment policy with the mechanisms for enforcing that policy, all inside the gateway.

The containment server is both a machine—typically a VM, for convenience—and a standard application server running on the machine. Conceptually, the combination of the gateway’s packet router and the containment server realizes a transparent application-layer proxy for all traffic entering and leaving the inmate network. The gateway’s packet router informs the containment server of all new flows to or from inmates, enabling it to issue *containment verdicts* on the flows. The router subsequently enforces these verdicts, which enable both *endpoint control* and *content control*. Endpoint control happens at the beginning of a flow’s lifespan and allows us to (i) forward flows unchanged, (ii) drop them entirely, (iii) reflect them to a sink server or alternative custom proxies, (iv) redirect them to a different target, or (v) rate-limit them. Once the gateway has established connectivity between the intended endpoints, it alone enforces endpoint control, conserving resources on the containment server. Content control, on the other hand, remains feasible throughout a flow’s lifespan and works by fully proxying the flow content through the containment server. We thus can (i) rewrite a flow, (ii) terminate it when it would normally continue still, or (iii) prolong it by injecting additional content into a flow that would otherwise already

terminate. Note that the connection’s destination need not necessarily exist: the containment server can simply *impersonate* one by creating response traffic as needed. Endpoint and content control need not mutually exclude each other; for example, it can make sense to redirect a flow to a different destination while also rewriting some of its contents. Figure 2 illustrates the flow manipulation modes. Note how the containment server remains passive during containment enforcement except in case of content rewriting (Figure 2(f)).

Besides flow-related containment management, the containment server also controls the inmates’ *life-cycle*. As the containment server witnesses all network-level activity of an inmate, it can react to the presence—and absence—of such *network events* using *activity triggers*. These triggers can terminate the inmate, reboot it, or revert it to a clean state for subsequent reinfection. For example, a typical life-cycle policy for a spambot might automatically restart the bot once it has ceased spamming activity for more than an hour. Another standard policy is to terminate an inmate that has begun to flood a particular recipient with more than a certain number of connection requests per minute.

5.5 Inmate control

To realize inmate life-cycle control, we require a component in the system that can create inmates, expire them, or adjust their life-cycle state by starting, stopping, or reverting them to a clean state. In GQ’s architecture, the containment servers issue life-cycle actions to an *inmate controller* located centrally on the gateway. Since we keep containment servers logically and physically separate from the gateway, we require a mechanism allowing them to communicate with the inmate controller. We re-

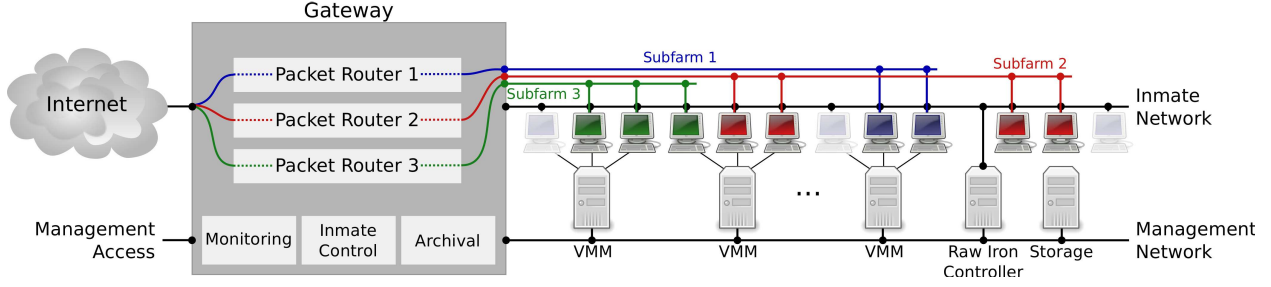


Figure 3: A possible subfarm scenario in GQ: three independent routers handle disjunct sets of VLAN IDs, thus enabling parallel experiments. For sake of clarity, we do not show per-subfarm infrastructure services individually.

alize this by maintaining an additional network interface only on the containment servers that allows them to interact directly, out-of-band of the inmate-related network activity, with the gateway. This controller understands the inmate hosting infrastructure and abstracts physical details of the inmates such as their hosting server and whether they run virtualized or on raw iron. The controller requires only the inmate’s VLAN ID in order to identify the target of a life-cycle action.

5.6 Subfarms

The combination of central gateway and inmate network parallelizes naturally: instead of a single packet forwarding logic handling the entire range of VLAN IDs on the inmate network, multiple instances of the packet router handle disjunct sets of VLAN IDs. This creates independent, self-contained, non-interfering habitats for different experiments. We call these habitats *subfarms*. Subfarms generally have their own containment server (thereby providing a subfarm-specific containment policy) and infrastructure services, but can vary widely in experiment-specific additional services. Shared network resources, as mentioned by Chen et al. [7], are possible, but we have not found them necessary or useful in practice—the very reason for creating independent subfarms is typically the need to configure systems differently or record separate data sets. Figure 3 illustrates subfarms.

5.7 Monitoring and packet trace archival

We use a two-pronged packet trace recording strategy. First, the packet routers record each subfarm’s activity from the inmate network’s perspective, i.e., with unroutable addresses for the inmates. This has the benefit of providing some extent of immediate anonymity in the packet traces, simplifying data sharing without the risk of accidentally leaking our global address ranges. Second, system-wide trace recording happens at the upstream interface to the outside network, thus capturing all activ-

ity globally and prior to any gateway-induced filtering or rewriting.

6 Implementation

6.1 Packet routing

We implement the subfarm’s packet routers on the gateway using the Click modular router [15]. We separate each subfarm’s Click configuration into a module containing invariant, reusable forwarding elements shared across all subfarms (400 lines) and a small (40 lines) configuration module that specifies each subfarm’s unique aspects, including the external IP address ranges, set of VLAN ID ranges from the inmate network to route, and logfile naming. The custom Click elements for the VLAN bridge, NAT, containment server flow handling, and trace recording comprise around 5000 lines of C++.

6.2 Containment server

We realize the containment server in Python, using a pre-forked, multi-threaded service model. The server logic comprises roughly 2200 lines, with 600 lines for the event trigger logic. The containment policies, including content rewriters, add up to 1000 lines.

Shimming protocol. To couple the gateway’s packet router to the containment server, we need a way to map/unmap arbitrary flows to/from the single address and port of the containment server. We realize this using a *shimming protocol* that shares conceptual similarity with the SOCKS protocol [14]: upon redirection to the containment server, the gateway injects a *containment request* shim message with flow meta-information into the flow. The containment server expects this meta-information and uses it to assign a containment policy to the flow. The containment server similarly conveys the containment verdict back to gateway using a *containment response* shim, which the packet router strips from the flow before relaying subsequent content back to the endpoint. For TCP, the shim injection and removal

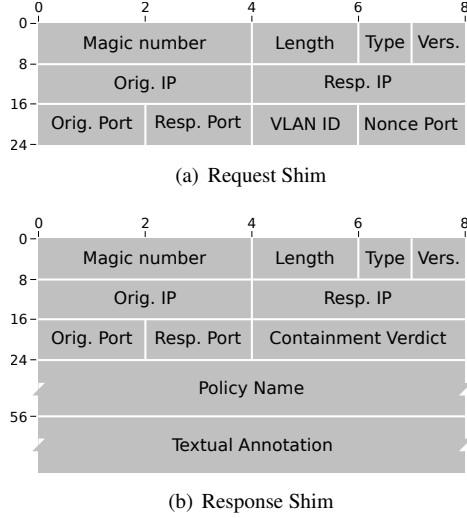


Figure 4: Shim protocol message structure.

requires bumping and unbumping of sequence and acknowledgement numbers; for UDP it requires padding the datagrams with the respective shims.¹ The request shim requires 24 bytes and begins with a preamble of 8 bytes containing a magic number (4 bytes), the message length (2 bytes), a message type indicator (1 byte), and a shim protocol version number (1 byte), followed by the original flow’s endpoint four-tuple ($2 \cdot 4$ bytes plus $2 \cdot 2$ bytes), the VLAN ID of the sending/receiving inmate (2 bytes) and a nonce port (2 bytes) on which the gateway will expect a possible subsequent outbound connection from the containment server, in case it needs to rewrite the flow continuously. The response shim can vary in length and requires at least 56 bytes, consisting of a similar preamble (8 bytes), the resulting endpoint four-tuple (12 bytes), the containment verdict (FORWARD, LIMIT, DROP, REDIRECT, REFLECT, or REWRITE as per Figure 2, possibly in feasible combination) as a numeric opcode (4 bytes), a name tag for the resulting containment policy (32 bytes) and a possible additional annotation string to clarify the context in which the containment server decided the containment verdict. Figure 4 summarizes the message structure.

An example of this containment procedure is shown in Figure 5. An inmate initiates the TCP handshake for an upcoming HTTP request (❶), which the gateway redirects to the containment server’s fixed address and port, synthesizing a full TCP handshake. Upon completion, the gateway injects into the TCP stream the containment request shim (❷). The containment server in turn sends a containment response shim (❸) including the containment verdict for the flow, in this case a REWRITE. To

serve as a transparent proxy rewriting the flow content, it also establishes a second TCP connection to the target via the gateway and the nonce port received as part of the containment request shim. The gateway forwards this TCP SYN to the target and relays the handshake between target and inmate (❹). The inmate completes its connection establishment and sends the HTTP request, which the gateway relays on to the containment server as part of the same connection that it used to exchange containment information, bumping the sequence number accordingly (❺). The containment server rewrites the request as needed (here changing the requested resource to another one) and forwards it on to the target, via the gateway (❻). The target’s response travels in the opposite direction and arrives at the containment server, which again rewrites it (here to create the illusion of a non-existing resource) and relays it back to the inmate (❼). (For brevity, we do not show the subsequent connection tear-downs in the figure.)

Policy structure. We codify containment policies in Python classes, which the containment server instantiates by keying on VLAN ID ranges and applies on a per-flow basis. We base endpoint control upon the flow’s four-tuple, and content control depends on the actual data sent in a given flow. Object-oriented implementation reuse and specialization lends itself well to the establishment of a hierarchy of containment policies. From a base class implementing a default-deny policy we derive classes for each endpoint control verdict, and from these specialize further, for example to a base class for spambots that reflects all outbound SMTP traffic. The containment server simply takes the name of the class implementing the applicable containment policy into the response shim (recall Figure 4(b)) in order to convey it to the gateway.

Configuration. The codified containment policies are customizable through a configuration file. This file serves four purposes: (i) it determines the initial assignment of a policy to a given inmate’s traffic, (ii) it typically specifies the individual or set of malware binaries with which we would like to infect a given inmate over the course of its life-cycles,² (iii) it specifies activity triggers (e.g., revert and reinfect the inmate once the containment server has observed no outbound activity for 30 minutes) and (iv) it specifies IP addresses and port numbers of infrastructure services in a subfarm (e.g., where to find a particular SMTP sink or HTTP proxy). Figure 6 shows a sample configuration snippet.

¹For large UDP datagrams this can introduce the need for fragmentation.

²As indicated earlier, we typically specify precisely which sample to infect an inmate with. However, GQ equally supports classic honeypot constellations in which dynamic circumstances (such as a web drive-by) determine the nature of the infection.

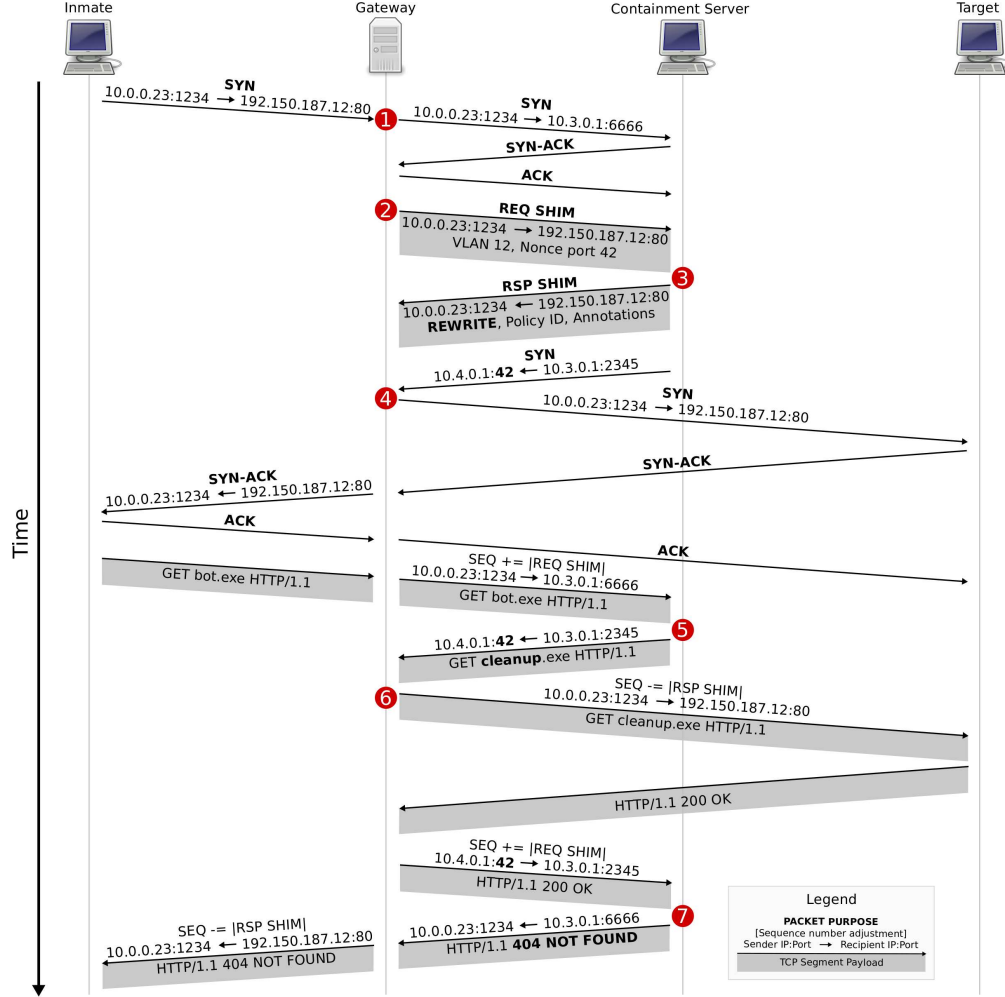


Figure 5: TCP packet flow through gateway and containment server in a REWRITE containment. See § 5.4 for details.

6.3 Inmate controller & sinks

We realize the inmate controller as a simple message receiver that interprets the life-cycle control instructions coming in from the containment servers. We use a simple text-based message format to convey the instructions. At startup, the controller scans the VMMs deployed on the management network to assemble an inventory of inmates and their VLAN IDs. We implement the controller in 470 lines of Python. The support scripts for managing inmate VMs and raw iron machines abstract concrete virtualization platforms (such as VMware’s ESX control scripts), in 2000 lines of shell code. We likewise implement GQ’s sink servers in Python. Our simplest *catch-all* server accepts arbitrary input and requires a mere 100 lines of code, while our most complex sink constitutes a fidelity-adjustable SMTP server that can grab greeting banners from the actual target and drop a configurable number of connections. It needs 800 lines, using the

same pre-forked multithreaded back-end as the containment server.

6.4 Raw iron management

Researchers have long known of the use of VM-detecting anti-forensics capabilities in malware toolkits [8]. Using a variety of techniques, the malware attempts to identify virtualized execution, which it perceives as a tell-tale sign of attempted forensic malware analysis. Rather than attempt to develop specific countermeasures (such as renaming interfaces or hiding other VM artifacts), GQ bypasses this problem by providing a group of identically configured small form-factor x86 systems running on a network-controlled power sequencer to enable remote, OS-independent reboots. Like our virtualized inmates, each raw iron system uses its own exclusive VLAN. From the gateway’s viewpoint, the only difference in the inmate hosting platform is the presence of an additional

```

[VLAN 16-17]
Decider = Rustock
Infection = rustock.100921.*.exe

[VLAN 18-19]
Decider = Grum
Infection = grum.100818.*.exe

[VLAN 16-19]
Trigger = *:25/tcp / 30min < 1 -> revert

[Autoinfect]
Address = 10.9.8.7
Port = 6543

[BannerSmtpSink]
Address = 10.3.1.4
Port = 2526

```

Figure 6: Example of a containment server configuration file: GQ will infect the inmates on VLAN IDs 16 and 17 iteratively with binaries from the `rustock.100921.*.exe` batch, using the “Rustock” containment policy. Similarly, the containment server will deploy Grum binaries on inmates with VLAN IDs 18 and 19 which contain the resulting infections accordingly. On all four VLAN IDs a lifecycle trigger reverts inmates to a clean state whenever the number of flows to TCP port 25 in 30 minutes hits zero. The last two sections specify the location in the subfarm of an auto-infection server and of an SMTP sink, respectively.

DHCP service and a separate reimaging process, executed by a dedicated raw iron controller.

The raw iron systems’ boot configuration alternates between booting over the network (leading to a subsequent OS image transfer and installation) and booting from local disk when network booting fails (leading to normal inmate execution). The raw iron controller has a network interface on a VLAN trunk which includes all raw iron VLAN IDs. The controller runs a separate Click configuration which creates a virtual network interface that multiplexes and demultiplexes traffic across the VLAN trunk, enabling unmodified Linux servers, including DHCP, TFTP, and NFS, to access the raw iron VLANs transparently.

To reimagine a system, we configure the controller’s DHCP server to send PXE boot information in reply to DHCP requests from the inmate. The power sequencer then reboots the inmate and the subsequent network boot installs a small Linux boot image. This image, once booted, downloads a compressed Windows image and writes it to disk using NTFS-aware imaging tools. Next, controller disables network-booting in the DHCP server and again power-cycles the raw iron inmate. The subsequent reboot yields a local booting of the locally installed OS image. This process takes around 6 minutes per reimaging cycle, sufficiently fast for hosting bots or other malware. We use a similar mechanism to capture disk images from a suitably configured OS image.

```

Inmate Activity
=====

Active subfarms: Botfarm, Clickbots

Subfarm 'Botfarm' [Containment server VLAN 11]
-----

Grum [xxx.yyy.0.170/10.3.9.241, VLAN 18]
-----
FORWARD
- C&C port          target      port      #flows
                   50.8.207.91.SteepHost.Net  http      682

REFLECT
- full SMTP containment target      port      #flows
                   *.*.*.*                smtp      144997

REWRITE
- autoinfection 6f007d640b3d5786a84dedf026c1507c
                   target      port      #flows
                   10.9.8.7      6543      6

SMTP sessions          93340
SMTP DATA transfers    93112

Rustock [xxx.yyy.0.164/10.3.9.247, VLAN 7]
-----
FORWARD
- C&C port          target      port      #flows
                   *.*.*.*                https     287

REFLECT
- simple SMTP containment target      port      #flows
                   *.*.*.*                smtp      280620

REWRITE
- C&C filtering      target      port      #flows
                   *.*.*.*                http      788

- autoinfection 0740eb0e408ea0b6cfa95981273f89bb
                   target      port      #flows
                   10.9.8.7      6543      21

SMTP sessions          182653
SMTP DATA transfers    284765

```

Figure 7: Section of actual report generated by the monitoring setup, for the “Botfarm” subfarm. GQ contains one inmate using the “Grum” policy, the other using the “Rustock” one. The REWRITES show that the containment server employs auto-infection (§ 6.6) for both, along with the MD5 hashes of the actual executables. The number of SMTP flows REFLECTed differs from the total number of SMTP sessions because we configured the SMTP sink to drop connections probabilistically. (Global inmate IP addresses anonymized.)

6.5 Reporting

We realize the reporting component using Bro [23]. We developed an analyzer for the shimming protocol to keep track of all containment activity on the inmate network, and track specific additional classes of traffic as needed (for example, we leverage Bro’s SMTP analyzer to track attempted and succeeding message delivery for our spambots). Bro’s log-rotation functionality then initiates activity reports on an hourly and daily basis. The reports break down activity by subfarm, inmate, and containment decision, allowing us to verify that the gateway

enforces these decisions as expected (for example, an unusual number of `FORWARD` verdicts might indicate a bug in the policy, and absence of any `C&C REWRITES` would indicate lack of botnet activity). We also pull in external information to help us verify containment (for example, we check all global IP addresses currently used by inmates against relevant IP blacklists). Figure 7 shows part of a report. The reporting setup extracts all of the per-inmate textual information from network activity.

6.6 Auto-infection & batch processing

Automating the infection (and re-infection, after a revert to clean state) of inmates is a key ingredient for saving us time operating GQ. We have implemented an auto-infection routine as follows. We configured auto-infection inmate master images to run a custom infection script at first boot (subsequent reboots should not trigger reinfection, as some malware intentionally triggers reboots itself). This script then contacts an HTTP server at a preconfigured address and port, requests the malware sample, and executes it, thus completing the infection. Note that we can realize the HTTP server as a `REWRITE` containment, simplifying the implementation substantially: the containment server observes the attempted HTTP connection anyway, and can thus proceed to impersonate the simple HTTP server needed to serve the infection. We realize this in a separate containment class that serves as a base class for all policies that operate using auto-infection. Again, VLAN IDs drive selection of a malware sample, as the configuration in Figure 6 illustrates. Processing batches of malware samples follows as a simple generalization: instead of serving the same sample repeatedly, we maintain the batch as a list of files and serve them sequentially.

6.7 External network address space

In practice we operate GQ using a set of five /24 networks. We use one network only to make our experiments' control infrastructure (at least each subfarm's containment servers, potentially also sink servers and other machinery) available externally, leaving four networks for our inmate population. The size of these allocations forms an upper bound on the number of inmates we can accommodate at any given time, but has so far proved sufficient: we experience no address tainting, i.e., the need to move around in our address range to preserve inmate functionality.

7 Operational Experiences

While containment policy development can easily feel like a chore, we have found that developing tight con-

tainment often forms a key ingredient for understanding the behavior of a sample under study. We now show examples of such insights learned during containment development.

Absence of activity. During our operation of GQ as a worm-capturing honeyfarm at the beginning of 2006, we captured 66 distinct worms belonging to 14 different malware families (according to classification by Symantec's anti-virus products at the time). Table 1 summarizes the captures. Note the high variation in incubation times: nine infection classes required more than three minutes on average for a propagation. This illustrates the importance of long-duration execution to observe the desired behavior, even in infections that supposedly act quickly.

Unexpected visitors. During our infiltration of the Storm botnet in 2008 [13, 17, 18] we developed sufficiently complete understanding of this particular malware family that a containment policy seemed to follow naturally. For the C&C-relaying proxy bots in the middle of the Storm hierarchy, we preserved outside reachability of the bots (the requirement for their becoming relay agents as opposed to spam-sourcing drones) and redirected all outgoing activity other than the HTTP-borne C&C protocol to our standard sink server. In June, we noticed apparent FTP connection attempts arriving at our sink server. Closer investigation revealed the attempted use of our Storm bots for iframe injection. An upstream botmaster used the bots' ability to receive and process SOCKS [14] message headers to initiate FTP connections to specific hosts, authenticate and log in with known credentials, download a specific HTML file and re-upload it with a malicious iframe tag inserted. At the time, articles on Storm frequently stated that its proxy bots did not themselves engage in malicious activity, and a correspondingly loose containment policy would have allowed these attacks to proceed unhindered.

Mysterious blacklisting. While deploying spambots of the Waledac family [26] in June 2009, we noticed to our great surprise that the Composite Blocking List (CBL) [6] listed our inmates—a strong indication of a possible containment failure. However, the only outside interaction we had permitted was a single test SMTP message, sent to a GMail server, because redirection to our default SMTP sink seemed to cause the bots to cede further activity. Subsequent independent testing revealed that the bots used a recognizable HELO greeting string (`wergvan`), that Google detected its presence, and that they informed blacklist providers of the IP addresses of SMTP speakers employing the string. This caused us to stop the policy of allowing even seemingly innocuous non-spam test SMTP exchanges.

Satisfying fidelity. It soon turned out that Waledac was not the only spambot family whose members paid close attention to the greeting banners served by the

EXECUTABLE	WORM NAME	# EVENTS/ # CONNS	INCUBATION PERIOD (S)
a####.exe	W32.Zotob.E	4 / 3	29.0
a####.exe	W32.Zotob.H	9 / 3	25.2
a####.exe	—	1 / 3	223.2
cpufuncntrl.exe	Backdoor.Sdbot	1 / 4	111.2
chkdisk32.exe	—	1 / 4	134.7
dllhost.exe	W32.Welchia.Worm	297 / 4 or 6	24.5
enbiei.exe	W32.Blaster.F.Worm	1 / 3	28.9
msblast.exe	W32.Balster.Worm	1 / 3	43.8
lsd	W32.Poxdar	11 / 8	32.4
NeroFil.EXE	W32.Spybot.Worm	1 / 5	237.5
sysmsn.exe	W32.Spybot.Worm	3 / 3	79.6
MsUpdaters.exe	W32.Spybot.Worm	1 / 5	57.0
RealPlayer.exe	W32.Spybot.Worm	2 / 5	95.4
WinTemp.exe	W32.Spybot.Worm	1 / 5	178.4
wins.exe	W32.Spybot.Worm	1 / 5	118.2
msnet.exe	W32.Spybot.Worm	1 / 7	189.4
msgupdates.exe	W32.Spybot.Worm	2 / 5	125.3
ntsf.exe	—	1 / 5	459.4
scardsvr32.exe	W32.Femot.Worm	4 / 3	46.2
scardsvr32.exe	W32.Femot.Worm	1 / 3	66.5
scardsvr32.exe	W32.Femot.Worm	55 / 3	96.6
scardsvr32.exe	W32.Femot.Worm	2 / 3	179.6
scardsvr32.exe	W32.Femot.Worm	1 / 5	49.3
scardsvr32.exe	W32.Femot.Worm	4 / 3	41.4
scardsvr32.exe	W32.Femot.Worm	1 / 3	41.1
scardsvr32.exe	W32.Valla.2048	1 / 5	32.2
scardsvr32.exe	—	1 / 7	54.8
scardsvr32.exe	W32.Pinfi	1 / 3	180.8
x.exe	W32.Korgo.Q	17 / 2	6.6
x.exe	W32.Korgo.T	7 / 2	9.5
x.exe	W32.Korgo.V	102 / 2	6.0
x.exe	W32.Korgo.W	31 / 2	5.9
x.exe	W32.Korgo.Z	20 / 2	6.6
x.exe	W32.Korgo.S	169 / 2	6.6
x.exe	W32.Korgo.S	15 / 2	8.6
x.exe	W32.Korgo.V	2 / 2	24.4
x.exe	W32.Licum	2 / 2	7.9
x.exe	W32.Korgo.S	3 / 2	10.4
x.exe	W32.Pinfi	1 / 2	329.7
x.exe	W32.Korgo.V	6 / 2	11.3
x.exe	W32.Pinfi	5 / 2	20.1
x.exe	W32.Pinfi	5 / 2	24.9
x.exe	W32.Pinfi	2 / 2	27.5
x.exe	W32.Korgo.V	1 / 2	27.5
x.exe	W32.Pinfi	1 / 2	63.1
x.exe	W32.Korgo.W	1 / 2	76.1
x.exe	W32.Korgo.S	1 / 2	18.0
x.exe	W32.Pinfi	1 / 2	58.2
x.exe	W32.Korgo.S	1 / 2	210.9
xxxx...x	Backdoor.Berbew.N	844 / 2	9.4
xxxx...x	W32.Info.A	34 / 2	7.2
xxxx...x	Trojan.Dropper	685 / 3	10.0
xxxx...x	W32.Pinfi	1 / 3	32.5
xxxx...x	W32.Pinfi	3 / 2	34.2
n/a	W32.Korgo.C	3 / 2	4.8
n/a	W32.Korgo.L	1 / 2	7.0
n/a	W32.Korgo.G	8 / 2	4.1
n/a	W32.Korgo.N	2 / 2	5.3
n/a	W32.Korgo.G	3 / 2	5.4
n/a	W32.Korgo.E	1 / 2	5.6
n/a	W32.Korgo.gen	1 / 2	5.0
n/a	W32.Korgo.I	15 / 2	4.3
n/a	W32.Korgo.I	1 / 2	5.4
multiple	W32.Muma.A	2 / 7	186.7
multiple	W32.Muma.B	2 / 7	208.9
multiple	BAT.Boohoo.Worm	1 / 72	384.9

Table 1: Self-propagating worms caught by GQ in early 2006. Events correspond to infections with the same executable. Next come the numbers of connections needed per infection to complete the propagation, and incubation times (the delay from initial infection in our farm to subsequent infection of the next inmate). We show delays of over 3 minutes in bold.

SMTP servers. As a consequence, we upgraded our SMTP sink to support *banner grabbing* for select connections: SMTP requests to a hitherto unseen host now caused the sink to actually connect out to the SMTP server and obtain the greeting message, relaying it back to the spambot. The more closely malware tracks whether the responding entity behaves as expected, the more likely we are to get drawn into an arms race of emulating and testing fidelity.

Protocol violations. On several occasions during our ongoing extraction of spam from spambots [17], our spam harvest accounting looked healthy at the connection level (since lots of connections ensued), but upon closer inspection meager at the content level (since for some bot families no actual message body transmission occurred). Closer investigation revealed that our SMTP sink protocol engine followed the SMTP specification [25] too closely, preventing the protocol state machine from ever reaching the DATA stage. The protocol discrepancies included such seemingly mundane details as repeated HELO/EHLO greetings or the format of email addresses in MAIL FROM and RCPT TO stanzas (with or without colons, with or without angle brackets).

Exploratory containment. Containment policy development need not always try to facilitate continuous malware operation. To better understand a sample, we frequently find it equally important to create specific environmental conditions under which the sample will exhibit new behavior. Before our infiltration of Storm, we tried to understand the meaning of the error codes returned in Storm’s delivery reports [17] using a dual approach of live experimentation, in which we exposed the samples to specific error conditions during SMTP transactions, and binary analysis. Interestingly, neither approach could individually explain all behavior, but by iterating alternation of the two—live experimentation confirming conjectures developed during binary analysis, and vice versa—we could solve the case. We used this approach again during our infiltration of the MegaD botnet [4]. Here, live experimentation allowed us to confirm the correct functionality of the extracted C&C protocol engine. During a recent investigation of clickbots [20] we used the approach to understand the precise HTTP context of some of the bots’ C&C requests.

Unclear phylogenies. When we take a set of specimen supposedly belonging to malware family X and subject it to a corresponding containment policy, we implicitly assume that the notion of a malware family X is valid and that samples belonging to family X will exhibit behavioral similarity. We find this assumption increasingly violated in practice. We have repeatedly noticed that third parties label samples inconsistently, and have even observed cases of split personalities in malware: in February 2010 we encountered a specimen that at times

showed MegaD’s C&C behavior, and at other times behaved like a member of the Grum family. While tight containment ensures that cause no harm when containment policy and observed behavior mismatch, it puts into question the idea of developing a library of containment policies from which we can easily pick the appropriate one. A batch-processing setup that enables some extent of automated family classification is thus an important tool to have. To this end, we reflect all outgoing network activity to our catch-all sink and apply network-level fingerprinting on the samples’ initial activity trace. We have successfully used this approach to classify over one million malware samples that we harvested from pay-per-install distribution servers [3].

7.1 Scalability

Traditional honeyfarms such as Potemkin [28] place strong emphasis on scalability of the inmate population: these systems quickly “flash-clone” new inmate to provide additional infectees as needed. For the worm model this made sense—it requires infectee chains to reliably identify infections, and the assumption that any incoming flow may constitute a potential worm infection implies the need to set aside resources potentially for each sender of traffic received in the honeyfarm’s IP address range. We feel today’s requirements for malware execution have shifted. Self-propagating malware infections are not extinct, but much less relevant and typically use entirely different vectors (such as the user’s activity on specific websites, as in the case of Koobface [27]). Correspondingly, careful filtering and resource management related to unsolicited, inbound traffic often remains a non-issue, particularly when considering the typical scenario of home-user machines deployed behind network address translators. We find it more important to achieve scalability in terms of providing independent experiments at varying stages of “production-readiness” (including development of the farm architecture itself) with moderate requirements of inmate population size, and convenient mechanisms for intentional, controlled infection of the inmate population. GQ’s architecture reflects these considerations.

The architecture we outlined in § 5 constrains scalability as follows. First, VLAN IDs are a limited resource. The IEEE 802.1Q standard limits the VLAN ID to twelve bits, allowing up to 4096 inmates. However, physical switches frequently support less. We can avoid this limitation naturally by moving to multiple inmate networks and prepending a gateway-internal network identifier to VLAN IDs for purposes of identifying inmates. Second, with a large number of inmates in a single subfarm, a single containment server becomes a bottleneck, as it has to interpose on all flows in the subfarm. We can defuse

the situation straightforwardly by moving to a cluster of containment servers, managed by the subfarm’s packet router. We would merely need to extend the router’s flow state tables entries to include an identifier of the containment server responsible for the flow. Several containment server selection policies come to mind, such as random selection under the constraint that the same containment server always handles the same inmate. Third, similarly to the containment server the central gateway itself becomes a bottleneck as the number of inmates and subfarms grows. To date, we have not found this problematic: the same 3Ghz quad-core Xeon machine with 5GB of memory that we deployed six years ago still serves us well today, running 5-6 subfarms in parallel with populations ranging from a handful to a dozen of inmates. Fourth, our globally routable IP address space is of limited size, creating an incentive to avoid blacklisting and leaking of our actively used addresses at all cost to reduce the rate at which we “burn through” this address space. Currently we find our address allocations sufficient. Should this change, we may opt to use GRE tunnels in order to connect additional routable address space available in other networks (provided by colleagues or interested third parties) to the system.

8 A Methodology for Developing Containment Policies

GQ currently does not by itself help to automate the development of containment policies. However, it provides the basis for a principled approach to containment policy development. When deploying new malware samples, we employ the following strategy. Beginning from a complete default-deny of interaction with the outside world, we execute the specimen in a subfarm providing a sink server that accepts arbitrary traffic without meaningfully responding to it, and to which the containment server reflects all outbound traffic. We thus understand the extent to which the specimen comes alive, and can inspect the nature of the attempted communication and in the best case infer the precise intent, or at least attempt to distinguish C&C traffic from malicious activity. We can then whitelist suspected-safe traffic for outside interaction, in the most narrow fashion possible. For example, we only allow an HTTP GET request, seemingly for C&C instructions, with similar URL path structure. Generally opening up HTTP would be overzealous, as malware might use HTTP both for C&C as well as a burst of SQL injections. We then iterate the process over repeated executions of the specimen until we arrive at a containment policy that allows just the C&C lifeline onto the Internet, while containing malicious activity inside GQ.

We fully acknowledge the manual and—depending on

the specimen at hand—time-consuming nature of this process, but crucially, GQ makes this process both explicit and feasible. We see substantial potential for facilitating automation. For example, leveraging recent results of automated C&C protocol extraction from binaries [4, 16] could aid in understanding the significance of individual flows. The language in which we express containment policies in the containment server forms another area of future work. The primary reason for our current use of Python is experience and convenience, but the general-purpose nature of the language complicates the creation of a tool-chain for processing policies. For example, a traffic generation tool that can automatically produce test cases for a given concrete containment policy would strengthen confidence in the policy’s correctness significantly. A more domain-specific, abstract language (like in Bro [23]) could simplify this.

It behooves us to contemplate the consequences of a containment arms race, in which malware authors explicitly try to defeat our approach to containment. It is easy to see—as mentioned by John et al. [12]—that botmasters can construct circumstances in which malware will cease to function as desired, despite our best efforts. For example, spam campaigners could scatter control groups of email addresses among the spam target lists and require successful delivery of messages to these addresses to keep a bot spamming. However, note the conceptual difference between achieving the desired goal of executing malware (say, longitudinal harvesting of spam messages) and tight containment preventing harm to others. While the former may fail, in our experience we can guarantee the latter, given tight containment. We are thus optimistic that we can operate malware safely to the point of attack and control traffic becoming so blended that we can no longer meaningfully distinguish them.

9 Conclusion

The GQ malware farm introduces containment as a first-order component in support of malware analysis. In building and operating GQ for six years, we have come to treat containment itself as a tool that can improve our understanding of malware under study. While modern malware increasingly resists longitudinal analysis in completely contained environments, GQ not only allows but encourages flexible and precise containment policies while maintaining acceptable safety. With ongoing development, we anticipate it continuing to provide core functionality for malware and botnet analysis well into the future.

References

- [1] P. Barford and M. Blodgett. Toward botnet mesocosms. In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.
- [2] U. Bayer, C. Kruegel, and E. Kirda. TTAalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.
- [3] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, USA, August 2011.
- [4] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM CCS*, pages 621–634, Chicago, IL, USA, November 2009.
- [5] J. Calvet, C. R. Davis, J. M. Fernandez, J.-Y. Marion, P.-L. St-Onge, W. Guizani, P.-M. Bureau, and A. Somayaji. The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet. In *Proceedings of the 26th ACSAC Conference*, pages 141–150, New York, NY, USA, 2010. ACM.
- [6] CBL. Composite Blocking List. <http://cbl.abuseat.org>, 2003.
- [7] J. Chen, J. McCullough, and A. C. Snoeren. Universal Honeyfarm Containment. Technical Report CS2007-0902, UCSD, September 2007.
- [8] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th Conference on Dependable Systems and Networks (DSN)*, pages 177–186. IEEE, 2008.
- [9] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a System to Catch Worms in a Quarter Million Places. Technical Report TR-06-004, International Computer Science Institute, September 2006.
- [10] A. W. Jackson, D. Lapsley, C. Jones, M. Zatkó, C. Golubitsky, and W. T. Strayer. SLINGbot: A System for Live Investigation of Next Generation Botnets. In *Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security*, pages 313–318, Washington, DC, USA, 2009. IEEE Computer Society.
- [11] X. Jiang and D. Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, page 2. USENIX Association, 2004.
- [12] J. John, A. Moshchuk, S. Gribble, and A. Krishnamurthy. Studying spamming botnets using Botlab. In *Proceedings of the 6th USENIX Symposium*

- on *Networked Systems Design and Implementation*, pages 291–306. USENIX Association, 2009.
- [13] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 3–14, Alexandria, Virginia, USA, October 2008.
 - [14] D. Koblas. SOCKS. In *Proceedings of the 3rd USENIX Security Symposium*. USENIX Association, September 1992.
 - [15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.
 - [16] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Symposium on Security and Privacy*, pages 29–44. IEEE, 2010.
 - [17] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the Spam Campaign Trail. In *Proceedings of the First USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, San Francisco, USA, April 2008.
 - [18] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamcraft: An inside look at spam campaign orchestration. In *Proceedings of the Second USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, Boston, USA, April 2009.
 - [19] McAfee Labs. Threats Report. <http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2010.pdf>, October 2010.
 - [20] B. Miller, P. Pearce, C. Grier, C. Kreibich, and V. Paxson. What’s Clicking What? Techniques and Innovations of Today’s Clickbots. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, volume Lecture Notes in Computer Science 6739. Springer, July 2011.
 - [21] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. T. Wroclawski, and S. Schwab. The DETER project: Advancing the science of cyber security experimentation and test. In *IEEE Intl. Conference on Technologies for Homeland Security (HST)*, page 7, November 2010.
 - [22] Norman. Norman SandBox. http://www.norman.com/security_center/security_tools/.
 - [23] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Proceedings of the 7th USENIX Security Symposium*, pages 31–51, 1998.
 - [24] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting Spam with Itself. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, March 2010.
 - [25] J. Postel. Simple Mail Transfer Protocol. RFC 821, August 1982.
 - [26] G. Tenebro. W32.Waledac Threat Analysis. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/W32_Waledac.pdf, 2009.
 - [27] N. Villeneuve. Koobface: Inside a Crimeware Network. <http://www.infowar-monitor.net/reports/iwm-koobface.pdf>, November 2010.
 - [28] M. Vrabie, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.
 - [29] Y. Wang, D. Beck, X. Jiang, and R. Roussev. Automated Web Patrol with Strider Honeymonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, March 2006.
 - [30] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security & Privacy*, pages 32–39, 2007.