THE UNIVERSITY OF TULSA
THE GRADUATE SCHOOL

A SYSTEM FOR SHARING ABUSE DATA WITH WEB HOSTING PROVIDERS

by
Matthew Weeden

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science
in the Discipline of Computer Science

The Graduate School
The University of Tulsa

2017

T H E U N I V E R S I T Y O F T U L S A

THE GRADUATE SCHOOL

A SYSTEM FOR SHARING ABUSE DATA

WITH WEB HOSTING PROVIDERS

by

Matthew Weeden

A THESIS

APPROVED FOR THE DISCIPLINE OF

COMPUTER SCIENCE

By Thesis Committee

_____, Chair

Tyler Moore

_____

Mauricio Papa

_____

Sal Aurigemma

ABSTRACT

Matthew Weeden (Master of Science in Computer Science)

A System for Sharing Abuse Data with Web Hosting Providers

Directed by Tyler Moore

61 pp., Chapter 4: Conclusion

(119 words)

This thesis presents a study on abuse notification to hosting providers and describes a system implemented to automate this notification process for the abuse data maintained by the StopBadware non-profit organization. We measured the effectiveness of sharing lists of URLs on Google's Safe Browsing blacklist with 41 hosting providers who requested those URLs that they host. We found that these reports had varying effects on hosting providers' ability to improve cleanup times and recompromise rates, though we found (through a matched pair analysis) that the reports yielded significant improvement in the remediation of the *reported* URLs. The implemented system furthers this work by providing an application program interface (API) that hosting providers can use to request these blacklisted URLs.

ACKNOWLEDGEMENTS

I would like to thank Dr. Tyler Moore for teaching and guiding me well and Marie Vasek for answering all of my tedious questions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Since the early 2000s, cybercriminals have used web-based malware to steal personal information, solicit purchases of counterfeit goods, and directly steal money from connected users. To accomplish their goals, many attackers leverage the credibility and popularity of legitimate websites by adding malicious code to the servers hosting these sites. What results is that many thousands of website owners are unknowingly serving Internet users malicious code.

Several actors in the private-sector have designed algorithms to automatically detect these compromised sites, which are amassed into blacklists. Blacklists can be used to warn search engine users of malicious sites, build firewalls that protect corporate networks, and automatically alert website owners of compromises.

StopBadware is a non-profit organization that facilitates a data sharing program for such blacklists. Website owners and hosting providers can visit `stopbadware.org/clearinghouse/search` to see the blacklisting history of their site or how many blacklisted websites are currently running on their network. Beyond this, hosting providers can email StopBadware to request a list of which URLs are currently being blacklisted that are hosted on their network.

This thesis presents, first, a study on the direct and long-term effects of this bulk URL sharing from 2009 to 2015. Then, we describe an implemented HTTP API that automates this sharing.

1.1 Prior Work

Much has been done to detect web-based malware of various forms as well as to evaluate the effectiveness of countermeasures. Provos et al. developed a mechanism to identify so-called "drive-by-downloads": websites that attempt to automatically download

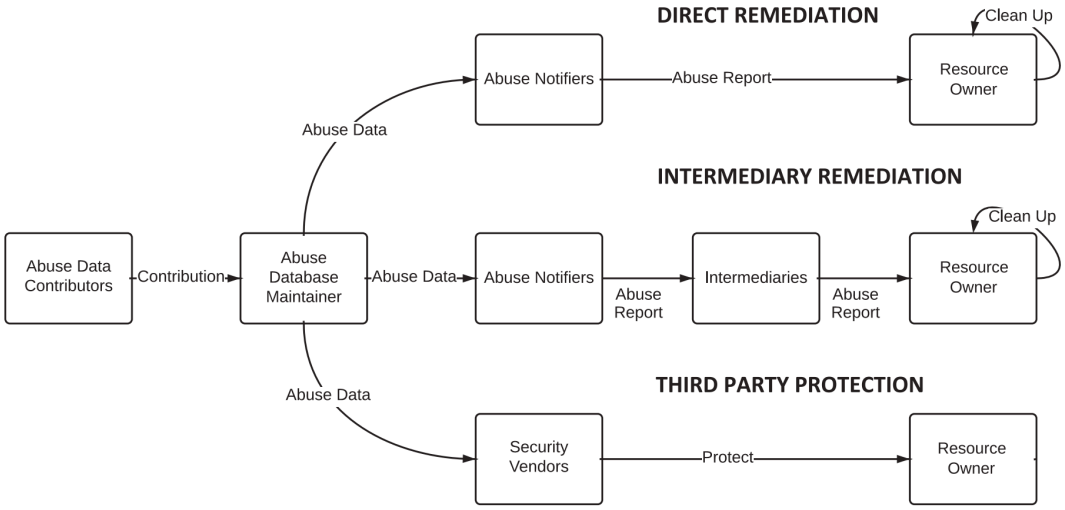

Figure 1.1: Abuse Reporting Infrastructure (courtesy [4]).

malware onto visitors' computers without any interaction [9]. This system ultimately became Google Safe Browsing, a real-time blacklist of websites crawled by Google that appear to be distributing malware. We use this Safe Browsing data in the study portion of the thesis, as described in Section 2.1.

Google itself uses the Safe Browsing service to notify webmasters of compromised sites, and Li et. al. analyzed data from Google on these interactions [7]. They found that notified, individual website operators who signed up for alerts from Google were more likely to clean up and do so more quickly. This notification is a central part of cleaning web-based malware because the actors who notice malicious web activity are often not in the position to remediate it. Jhaveri et al. constructed a model of the abuse reporting infrastructure shown in Figure 1.1 [4]. In this thesis, we study the intermediary remediation branch of reporting where Google Safe Browsing is the abuse data contributor/maintainer, StopBadware is an abuse database maintainer/notifier, and hosting providers are intermediaries.

A key motivation for this thesis is that intermediaries are often well positioned to intervene and assist in malware remediation, but that more work is needed to make it easier for them to become aware of issues so that they can help. One challenge has been the difficulty in identifying who is responsible for what in the varied industry of web hosting. Tajalizadehkhoob et al. presented an effective means of identifying hosting providers that

can mitigate that challenge [12]. The API described in this thesis presents another solution to bring down the costs of combating cybercrime for intermediaries.

Despite the many challenges, a number of researchers have been investigating the effectiveness of sharing abuse data with hosting providers. Vasek and Moore conducted an experiment in which they reported web-based malware infections to two entities: hosting provider and either webmaster or registrar [13]. They found that detailed abuse reports that articulated the website's malicious activity were effective, but that reports lacking such details were indistinguishable from doing nothing at all. In a follow-up study using a dataset of URLs used by the Asprox botnet, Cetin et al. confirmed that detailed reports shorten the time required to cleanup [2]. They also found no evidence that the email address of the abuse report sender affected the time to clean. Nappa et al reported 19 malware exploit servers to hosting provider contacts [8]. Canali et. al. set up websites on 22 shared hosting providers and attacked them [1]. After 25 days of attack, they sent out abuse notifications to the hosting providers and measured the responses. They found that most of their compromises were never remediated fully.

A few other studies have transmitted vulnerability reports to intermediaries and website operators. Dumeric et. al. reported susceptibility to the Heartbleed OpenSSL vulnerability to the hosting provider or internet service provider contact [3]. Notified hosts increased the patching rate by 47%. Kührer et. al. notified operators of NTP servers vulnerable to DDoS amplification attacks, observing a drop from 1.6 million vulnerable hosts to 126 000 in just three months [5]. Li et. al. ran a study notifying different responsible parties (hosting provider WHOIS contacts, national CERTs) about three types of vulnerabilities and found that sending detailed notices to hosting provider WHOIS contacts was the most effective [6]. Stock et. al. notified different responsible parties (varying webmaster, host, and country-level contacts) about WordPress and client-side XSS vulnerabilities [11]. They found a small statistical effect of all their notification efforts, while despairing over the inefficacy of large-scale notification campaigns.

The work presented in Chapter 2 differs in a number of ways. Much of the prior work has shared abuse reports for a single URL with the operators of websites. By contrast, we study the effect of reporting web-based malware URLs in bulk to a single requesting

intermediary. Furthermore, our dataset spans a much longer period of time (6 years). With one exception [7], in the prior work the abuse reports are sent unsolicited, while the study in Chapter 2 reports to requesting hosting providers. We also consider *compromises* rather than *vulnerabilities* (which may or may not later become compromised) making the incentive to clean much higher. Finally, unlike the other studies, we do not provide the intermediaries resources to help guide the cleanup. This is because for bulk URL sharing, there can be significant heterogeneity in the types of malware and the ensuing cleanup strategy. Hence, Chapter 2 complements the prior work by investigating abuse data sharing in a new context.

The system described in Chapter 3 is an attempt to further reduce the cost of sharing malware data with hosting providers. The system serves hosting providers specifically and will send clients all the blacklisted URLs hosted on their network, not just newly-blacklisted URLs as Google's own tools for hosting providers do. This is especially useful for notifying hosting providers of long-lived compromises.

Of course, malware URLs are not the only cybercrime data that can be shared. For example, Spamhaus Technology also provides a datafeed service to intermediaries (ISPs and corporate networks in this case) [10]. Administrators can use these feeds of IP and domain name blacklists to decide whether to accept incoming email to their networks. Our system's aim is to support remediation of malicious activity, not only protection against it.

1.2 Structure and contribution of this thesis

1.2.1 Thesis Statement

We empirically investigate the effectiveness of bulk abuse reports sent to requesting hosting providers. Based on the observed success of the approach, we design and implement an API to enable ongoing sharing for the StopBadware Clearinghouse.

1.2.2 Structure and Contribution

Chapter 2 describes a study in which we contribute to and investigate the remediation of web-based malware by hosting providers. Specifically, we measure the direct and long-term effects of notifying hosting providers of the URLs hosted on their network that are blacklisted by Google's Safe Browsing service. To make these measurements, we look

at the volume of URLs on the blacklist belonging to the hosting providers before and after we shared with them.

Chapter 3 presents a system for automating this bulk sharing of blacklisted URLs. We describe the requirements for this system then give architectural details of the implementation. The appendices include a user manual for hosting providers who register for the service, a maintenance manual, and the system's source code.

CHAPTER 2

MEASURING THE IMPACT OF SHARING ABUSE DATA WITH WEB HOSTING PROVIDERS

Many hosting companies have dedicated technical staff who assist in the cleanup of customer websites infected with malware, but their efficacy can vary greatly. Some merely forward received reports to their customers. Some will reach out to blacklist operators directly, who might offer some general instructions for cleanup and the ability to request removal once the infection has been eradicated. Others have reached out to StopBadware, a non-profit anti-malware organization established in 2006. In addition to helping individual website owners who have been hacked, StopBadware occasionally shares lists of malware URLs to inquiring hosting providers. This data is not made publicly available because of the data agreements StopBadware has with its providers. In this chapter, we empirically examine what happens to organizations after they request such data from StopBadware.[1]

We make the following contributions:

1. We report on an observational study where 41 web hosting providers received bulk reports from StopBadware on 28 548 URLs flagged as malicious by Google Safe Browsing between 2010 and 2015.

2. Directly sharing URLs with hosting providers is an effective strategy for cleaning up those particular URLs. We find that the median report to clean time for shared URLs is 101 days, compared to 153 days for a comparative sample of hosting providers that were not notified. Furthermore, the recompromise rate for shared URLs is 4%, compared to 10% for similar providers who did not receive information from StopBadware.

[1]The contents of this chapter were published in [14].

3. We observe a positive correlation between the time a URL has been blacklisted and the time required to clean it once reported. Long-lived malware URLs prove difficult to clean even after a provider is notified.

4. We find that for some providers, the long-term impact of sharing data is positive: 39% of providers demonstrably reduced the time from blacklisting to remediate malware in time periods after a malware report. But for 52%, no long-term improvement was observed, and for 9%, the response time actually got worse.

5. We find that providers that improved after receiving a report cleaned subsequent infections two months faster on average. Providers whose performance worsened after sharing cleaned URLs six months slower. Furthermore, worsened providers had, on average, more than eight times more recompromised URLs compared to providers that either improved or remained steady.

In 2.1, we describe the methodology for collecting our data. In 2.2, we describe the results of our study, starting with the direct impact of sharing in 2.2.1, the long-term impact in 2.2.2, then the results a hosting provider matched pair analysis in 2.2.3. Section 2.3 gives some concluding remarks to the study.

2.1 Data Collection Methodology

2.1.1 Inquiries to StopBadware

StopBadware runs a manual, independent reviews process for three data providers: Google Safe Browsing, ThreatTrack Security, and NSFocus. When a webmaster searches for their URL through StopBadware[2], they can find all the times that the URL was blacklisted by any partner, and if it is currently blacklisted, they can request a review. Users can also search for Internet Protocol (IP) addresses and Autonomous System numbers (ASNs). Searching for IP addresses and ASN lists the number of URLs in any blacklist hosted and prompts the user, if responsible for that IP or ASN, to request a bulk data dump from StopBadware.

[2] `https://www.stopbadware.org/clearinghouse/search`

Note that by URL, we mean blacklisting entries. For example, perhaps malware was found on `verybad.example.com/nogood/evil.php`. This could potentially be the blacklisting entry. Other potential entries could be `example.com`, `verybad.example.com`, `example.com/nogood/`, or `verybad.example.com/nogood/`[3]. We refer to these blacklisting entries as URLs moving forward.

If the user is indeed responsible for that hosting space, URL information is shared as per agreements with the data partners. StopBadware shares as much data with hosting providers as their partners allow. For instance, Google Safe Browsing only permitted StopBadware to share a limited amount of data. In most cases, reports include data from at least two providers. All reports include specific URLs blacklisted and some include additional IP information. No further information is shared regarding particular compromises. However, some hosting providers later followed up on specific compromise by making requests to StopBadware's independent review process.

For the purposes of this chapter, we only investigate URLs blacklisted by Google Safe Browsing, even though StopBadware also shared URLs flagged by the other sources. We elected to do this because Google's blacklist contains relatively homogeneous malware URLs used to facilitate drive-by downloads, whereas their other partners' blacklists offer more heterogeneous lists of multiple forms of bad activity. These lists sometimes include content that is not necessarily malware or universally recognized as such. Consequently, for some URLs on these other lists a slow removal may sometimes reflect a policy decision as opposed to a hampered response. By sticking to Google reports for our study, we can measure the time to clean up a consistent, universally undesirable form of online wickedness.

We note that Google Safe Browsing also offers alerts for network operators of recently blacklisted URLs[4]. These alert network operators of new compromises on their network, whereas the reports in this study report **all** known compromises on their network.

From 2010 through 2015, StopBadware received 88 requests for bulk data from 69 different stakeholders ranging from country CERTs (Computer Emergency Readiness Teams) to AS (Autonomous System) owners to free domain services. For the purpose of this

[3] `https://developers.google.com/safe-browsing/v4/urls-hashing#suffixprefix-expressions`
[4] `http://www.google.com/safebrowsing/alerts/`

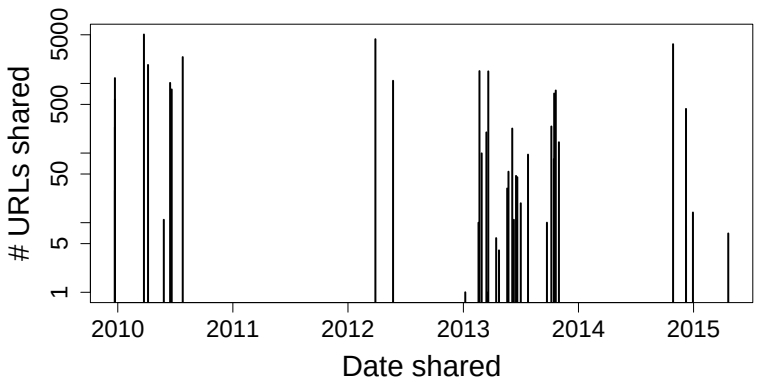

Figure 2.1: Bulk data reports sent from StopBadware to requesting hosting providers over time.

study, we wanted a more homogeneous group of entities to study; thus, we only considered hosting providers large enough to own their own AS, leaving us with 55 requests. A number of entities had multiple requests; removing those resulted in 42 remaining requests. Finally, we removed one AS whose request came a few weeks prior to the end of our study, since we could not reliably measure the long-term impact of sharing. Note that we refer to ASes and hosting providers interchangeably in the subsequent analysis. Figure 2.1 plots when StopBadware shared URLs with hosting providers over time. Each line indicates the number of URLs shared for each request.

2.1.2 Defining Malware Cleanup

We do not directly test websites for the presence of malware. Instead, we rely upon the outside judgments of blacklist providers to assess when a website is compromised, and therefore also when it becomes cleaned. The blacklists used by StopBadware are dynamic, and the operators strive to remove websites from the list as soon as they are believed to be clean. While we are not aware of any published studies of the accuracy of such blacklists, it is widely believed that these lists have very low false positive rates and modest false negative rates.

In most circumstances, it is straightforward to determine when a website is clean based on our data: it is simply the time that the website comes off the blacklist. Yet it is much less clear cut for a significant minority of websites that are placed back onto the blacklist shortly after being marked clean. In the extreme case, 0.05% of websites come on

and off the blacklist 10 times or more. Some URLs rejoin the blacklist after a few hours, while others return years later.

Re-blacklisting within a short period of time from the initial compromise could demonstrate that the attackers used the same vector of compromise or exploited a backdoor they left behind. It may also signal that malware managed to temporarily evade detection. To a first approximation, such websites have never really been cleaned. To that end, we attempted to distinguish between re-blacklisting events that signaled that the URL was never fully cleaned up and re-blacklisting events that signal that the URL was cleaned up and then was later compromised.

We check blacklist updates hourly. We consider a URL to be *clean* if it has been off the blacklist and stayed off for 21 days. If a URL is blacklisted after that 21-day period, we consider it to then be *recompromised*. If a URL falls of the blacklist and rejoins within 21 days, we consider the compromise to never have been cleaned up.

2.1.3 Measuring Secure Outcomes

We want to study both the direct impact of sharing URLs with the hosting providers and the indirect, longer term impact of sharing. To measure the direct impact of sharing, we looked at the time from when we reported the URLs to the time they were cleaned up, which we refer to as the *report-to-clean* time. Because many of the URLs shared had been compromised for a very long time before the hosting provider contacted StopBadware, measuring from the time of compromise to cleanup would not accurately measure provider effort compared to the report-to-clean time.

To measure the indirect impact of sharing, we compare hosting provider performance in the period before StopBadware shared URLs to the period afterwards. The idea is that some providers, upon receiving information about compromised websites in their network, will make improvements to the detection and remediation process that benefits their long-run security.

For measuring both the direct and indirect impact of sharing, we use survival analysis, a technique that works with *censored* data. Survival analysis allows us to include all data

points, even the URLs reported which never came off the blacklist during the measurement interval.

We assign all URLs blacklisted before the reporting date to the *pre-contact* group. These are then compared with URLs blacklisted after the reporting date in the *post-contact* group. For both groups, we compute the *blacklist-to-clean* time, that is, the time from when each URL is added to the blacklist to the time it is marked clean. We censor the URLs blacklisted on the reporting date (since any still compromised at that time were shared with hosting providers). To make these time periods more comparable, we only consider URLs blacklisted within two years of the report date. If we sent the report in the last two years, we consider the same length of time before and after the report. For the post-contact group, any URLs still compromised at the end of the period are also censored.

In addition to survival probability, we also compute the overall recompromise rates for the pre- and post-contact groups. Recompromise rates, aggregated over a hosting provider during an extended period of time, offer a good indication of how effective the provider's efforts to clean up compromised websites are.

2.2 Results

2.2.1 Direct impact of sharing

We first look at the rate at which URLs that StopBadware directly shared with hosting providers come off the blacklist. Figure 2.2 plots the survival probability for all 28 541 reported URLs. We find that within two weeks of receiving a report from StopBadware, about 20% of URLs come off the blacklist. Half the URLs come off the blacklist within 100 days of the report. Nonetheless, a significant minority of reported URLs remain compromised long after sharing. Approximately 40% remain blacklisted one year after StopBadware reported the URLs to the hosting provider.

We observe significant variation in the report to clean times by hosting provider. Figure 2.3 plots the survival probabilities for individual hosting providers (solid black line), along with the overall survival probability from Figure 2.2 in dashed lines. Note that we include only the 33 providers that experienced at least 10 malware reports both before and

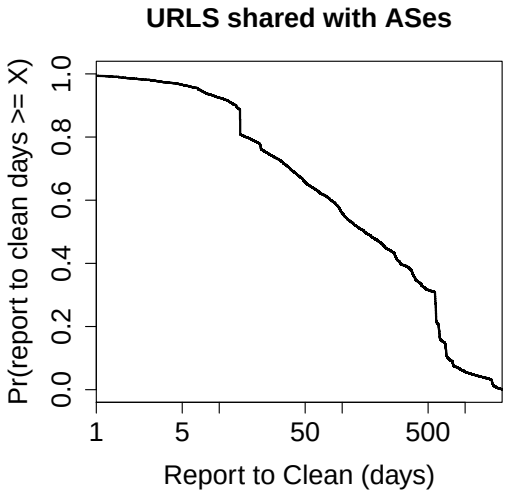

Figure 2.2: Survival probability for the number of days from a bulk report to clean (*reported* URLs)

after sharing occurred. Mult indicates that the AS received multiple bulk reports from StopBadware.

Summary statistics for these providers is also provided in Table 2.1. The median number of total compromised URLs for the hosting providers receiving reports was 2 212, whereas the median number shared by StopBadware is 225, approximately 10% of the total. We note substantial variation in the number of malware URLs shared, ranging from only 1 to over 5 000.

Some providers clearly reacted more quickly to the information provided by StopBadware than others. For example, providers 2, 3, and 4 clearly outperformed the rest, cleaning up more quickly than the other providers. By contrast, providers 8 and 9 lagged substantially, waiting over a year to clean up the vast majority of URLs shared. For many other providers, the differences were not so clear cut. For instance, provider 22 lagged for the first couple months, but then cleaned up most URLs and eliminated most long-lived infections at above average speed.

As noted in Section 2.1.3, shared URLs have a longer lifespan than other URLs. This might be because they are more likely to be maliciously registered than compromised and attackers are good at hiding. This could also be from particularly pernicious bits of

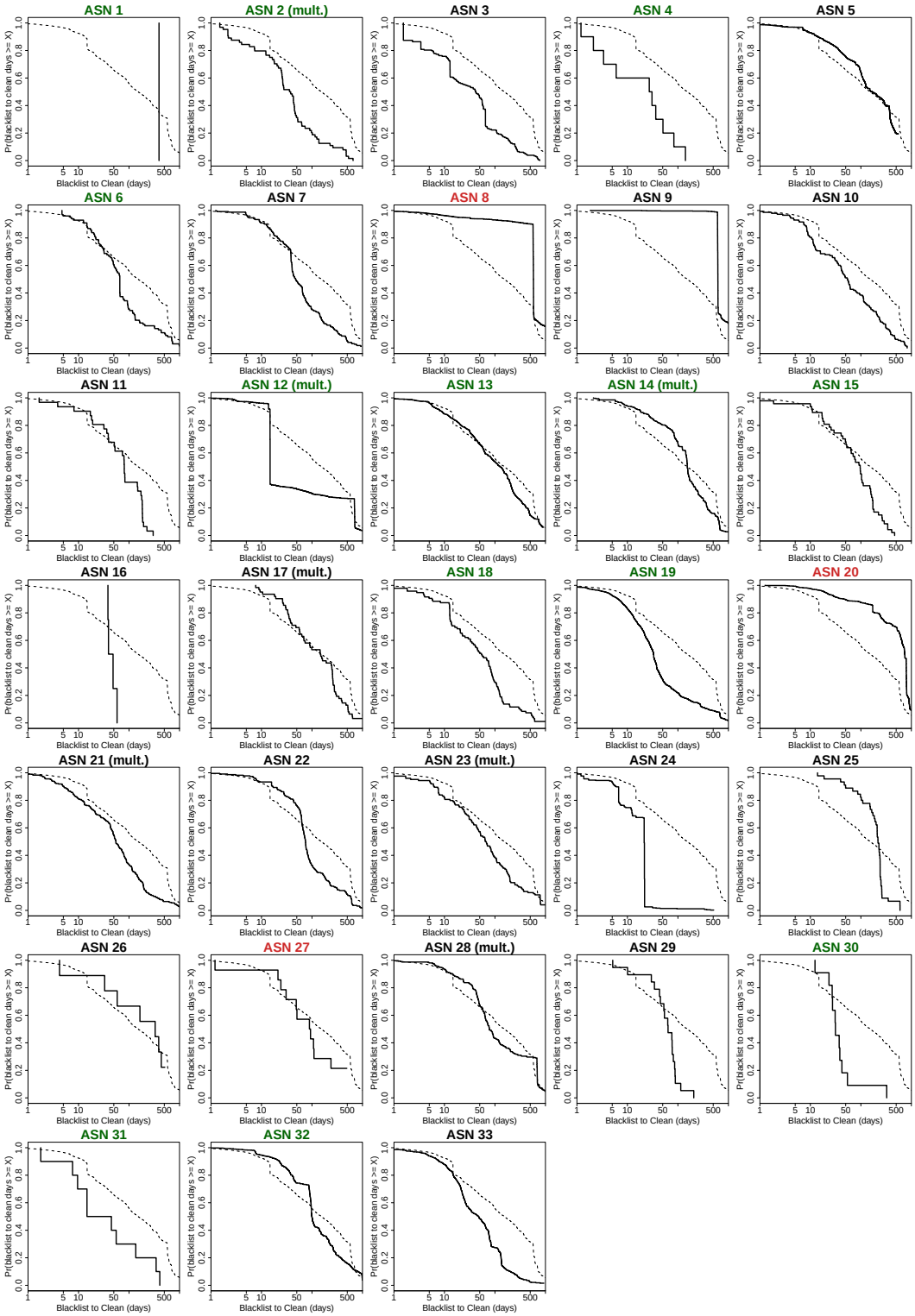

Figure 2.3: Survival probability for the time from reporting to clean per hosting provider. Figures are titled green if the hosting provider improved after contact, and red if they worsened. Dotted line indicates the average.

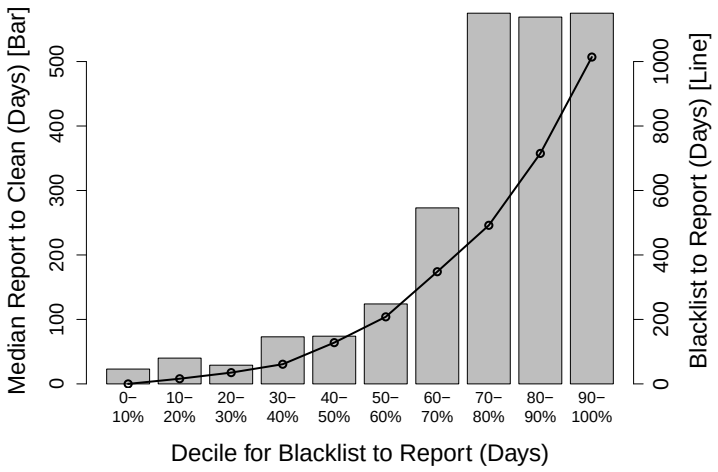

Figure 2.4: Measuring long lived malware: comparing report to clean times (bar) with blacklist to report times (line) for all reported URLs.

malware. While measuring the report-to-clean time mitigates this effect to some degree, long-lived malware is nonetheless harder to clean. We now attempt to quantify this effect.

Figure 2.4 demonstrates the relationship between long-lived malware URLs and the time required to clean them after reporting. The line plots the number of days from blacklist to when the report is sent by decile. For example, 20% of reported URLs had been blacklisted for 35 days or less when they were shared, but 60% had been blacklisted for 348 days or less. Meanwhile, 10% of shared URLs had already been on the blacklist for at least 1 014 days when StopBadware shared them. The bar plot indicates the median number of days from reporting to cleanup for each decile of the blacklist to report. The plot clearly demonstrates a strong positive relationship between long-lived infections and a slower time to clean once reported.

We note that one-off bulk reports like those shared between StopBadware and hosting providers are most helpful if they reflect a more systematic strategy to reduce abuse on a network, rather than a temporary fix. Thus, we spend the rest of this chapter analyzing the lasting effects of bulk reports by studying what happens to URLs reported after sharing has taken place.

		Direct Reporting			Pre-Contact			Post-Contact			Long-Term Impact		
ASN	# URLs	# Shared	Day Shared	Report to Clean	# URLs	Blacklist to Clean	Recomp. Rate	# URLs	Blacklist to Clean	Recomp. Rate	Δ Blacklist to Clean	Δ Recomp. Rate	Survival Prob.
1	91	1	2013-03-20	393 days	12	46 days	0	34	14 days	0.029	31.5 days	-0.029	improved
2	576	82	2013-10-15	88 days	282	112 days	0.103	179	39 days	0.039	73.2 days	0.064	improved
3	3 740	198	2013-03-15	89 days	532	56 days	0.041	231	44 days	0.087	12.3 days	-0.045	unclear
4	44	10	2013-09-23	38 days	23	35 days	0.087	16	34 days	0.188	0.9 days	-0.101	improved
5	93 102	3 664	2014-10-28	247 days	43 061	52 days	0.061	11 325	69 days	0.032	-17 days	0.03	unclear
6	2 730	99	2013-02-28	151 days	353	99 days	0.142	2164	50 days	0.081	49.3 days	0.06	improved
7	2 663	241	2013-10-07	125 days	1 239	92 days	0.084	1 278	91 days	0.058	0.6 days	0.026	unclear
8	21 984	2 392	2010-07-26	696 days	6 616	169 days	0.114	2 457	288 days	0.073	-119.1 days	0.041	worsened
9	3 980	1 018	2010-06-16	799 days	456	322 days	0.094	976	334 days	0.04	-12 days	0.054	unclear
10	1 797	143	2013-10-31	143 days	866	89 days	0.07	421	80 days	0.076	8.6 days	-0.006	unclear
11	58	31	2013-05-20	102 days	33	81 days	0	25	76 days	0.04	5.5 days	-0.04	unclear
12	9 679	5 046	2010-03-25	244 days	5 577	122 days	0.042	2636	52 days	0.039	70.2 days	0.003	improved
13	13 383	792	2013-10-21	232 days	5 207	88 days	0.083	4 522	63 days	0.028	24.4 days	0.055	improved
14	26 716	4 322	2012-03-28	242 days	2 077	132 days	0.087	12 012	56 days	0.13	76.2 days	-0.043	improved
15	454	47	2013-06-17	121 days	151	84 days	0.066	155	53 days	0.071	31.6 days	-0.005	improved
16	76	4	2013-04-24	46 days	11	149 days	0	47	15 days	0	133.2 days	0	unclear
17	698	234	2009-12-23	226 days	500	121 days	0.036	60	120 days	0.1	1.5 days	-0.064	unclear
18	624	95	2013-07-25	110 days	261	123 days	0.111	241	68 days	0.095	54.7 days	0.016	improved
19	33 909	1 491	2013-03-21	122 days	12 924	239 days	0.077	16 323	152 days	0.039	86.6 days	0.038	improved
20	2 212	820	2010-06-21	650 days	725	164 days	0.113	185	509 days	0.054	-344.9 days	0.059	worsened
21	3 739	1 193	2009-12-23	149 days	1 403	155 days	0.066	413	86 days	0.099	68.7 days	-0.034	unclear
22	3 927	225	2013-06-05	167 days	812	72 days	0.087	266	68 days	0.064	3.6 days	0.024	unclear
23	1 554	590	2009-12-23	207 days	530	78 days	0.066	274	62 days	0.088	15.9 days	-0.022	unclear
24	2 294	431	2014-12-08	22 days	1 282	101 days	0.035	293	114 days	0.02	-13.3 days	0.015	unclear
25	317	45	2013-06-21	214 days	82	91 days	0.146	43	71 days	0.047	19.6 days	0.1	unclear
26	186	9	2014-12-08	271 days	85	111 days	0.141	48	110 days	0 days	0.9 days	0.141	unclear
27	69	14	2014-12-30	163 days	45	58 days	0.156	15	121 days	0	-63.6 days	0.156	worsened
28	16 057	1 844	2010-04-07	296 days	3 596	186 days	0.058	1 102	193 days	0.089	-7 days	-0.031	unclear
29	617	19	2013-07-02	68 days	124	51 days	0.032	288	33 days	0.038	18.6 days	-0.006	unclear
30	249	11	2010-05-27	59 days	134	126 days	0.067	19	17 days	0.158	108.9 days	-0.091	improved
31	318	10	2013-02-18	104 days	29	96 days	0.172	216	27 days	0.056	68.7 days	0.117	improved
32	105 586	1 092	2012-05-23	248 days	9 252	141 days	0.094	51 504	60 days	0.044	80.6 days	0.05	improved
33	6 688	717	2013-10-16	93 days	3 344	58 days	0.051	2871	70 days	0.055	-12.5 days	-0.004	unclear
					*M*edian values for the 33 providers listed above:								
	2 212	225		151 days	530	99 days	0.077	274	68 days	0.055	15.9 days	0.015	

Table 2.1: Summary statistics for 33 hosting providers who had at least 10 URLs blacklisted before and after StopBadware shared data.

2.2.2 Long-term impact of sharing

We want to find if sending bulk reports to hosting providers creates a lasting impact on efforts to counter abuse. To this end, we consider all the hosting providers we report to. We then compare the URLs added to the blacklist before we shared with the hosting provider (pre-contact) with all the URLs added to the blacklist after (post-contact), as described in Section 2.1.

Figure 2.5 compares the cleanup rate for all ASes we reported to before and after we contacted them. We notice that cleanup is slightly faster for the post-contact URLs – about 80% of URLs are cleaned up within 100 days whereas only about 70% of URLs before StopBadware contact are cleaned up within 100 days.

Despite the overall improvement, significant variation exists between individual hosting providers. We now look further into the effects of sharing on each individual AS. Figure 2.6 plots the per-AS survival plots, while Table 2.1 reports summary statistics split

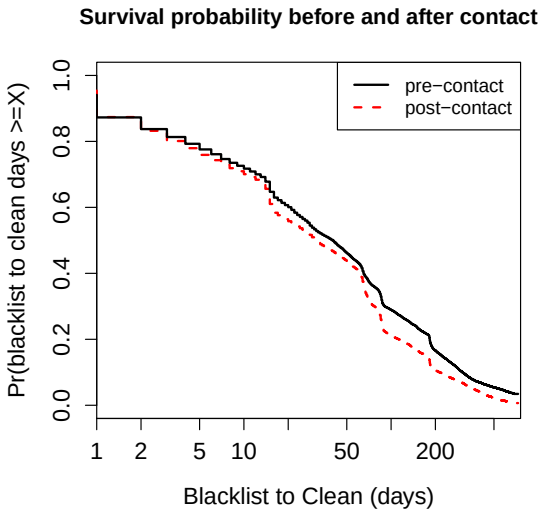

Figure 2.5: Survival probability for the number of days from blacklisting to clean for URLs pre- and post-contacting the host (all reported-to hosting providers).

between pre- and post-contact. As before, we only include survival plots for the 33 ASes we reported to who also have at least 10 URLs blacklisted both before and after contact.

We notice that AS 12 took 15 days on average to clean URLs from the time they were blacklisted, good for the shortest time in our study. This is especially impressive given the provider's relatively large size (nearly 10k malware URLs observed). Before StopBadware shared URLs with AS 12, they averaged 122 days from blacklist to clean and 4.2% of cleaned URLs were later recompromised. However, after sharing, blacklist-to-clean time improved to 52 days. We note we shared URLs with AS 12 in 2010; we hypothesize the time of sharing could affect the response. By contrast, AS 20 did not improve their cleanup after sharing with them – they had an average of 509 days from blacklist to clean and were the worst AS in our study. We also shared data with AS 20 in 2010. The blacklist to clean time was larger after our report. However, the recompromise rate decreased by 6 percentage points after sharing data.

Looking at each AS individually shows the heterogeneity in the efficacy of reporting and the dimensions of what makes an effective cleanup strategy. ASes like AS 1 took fewer days to clean a URL (from blacklist time) after reporting, but had a higher recompromise rate. ASes like AS 8 took significantly more days after reporting, but had a lower recompromise rate. On one hand, it is quicker to clean a URL if additional time is not spent

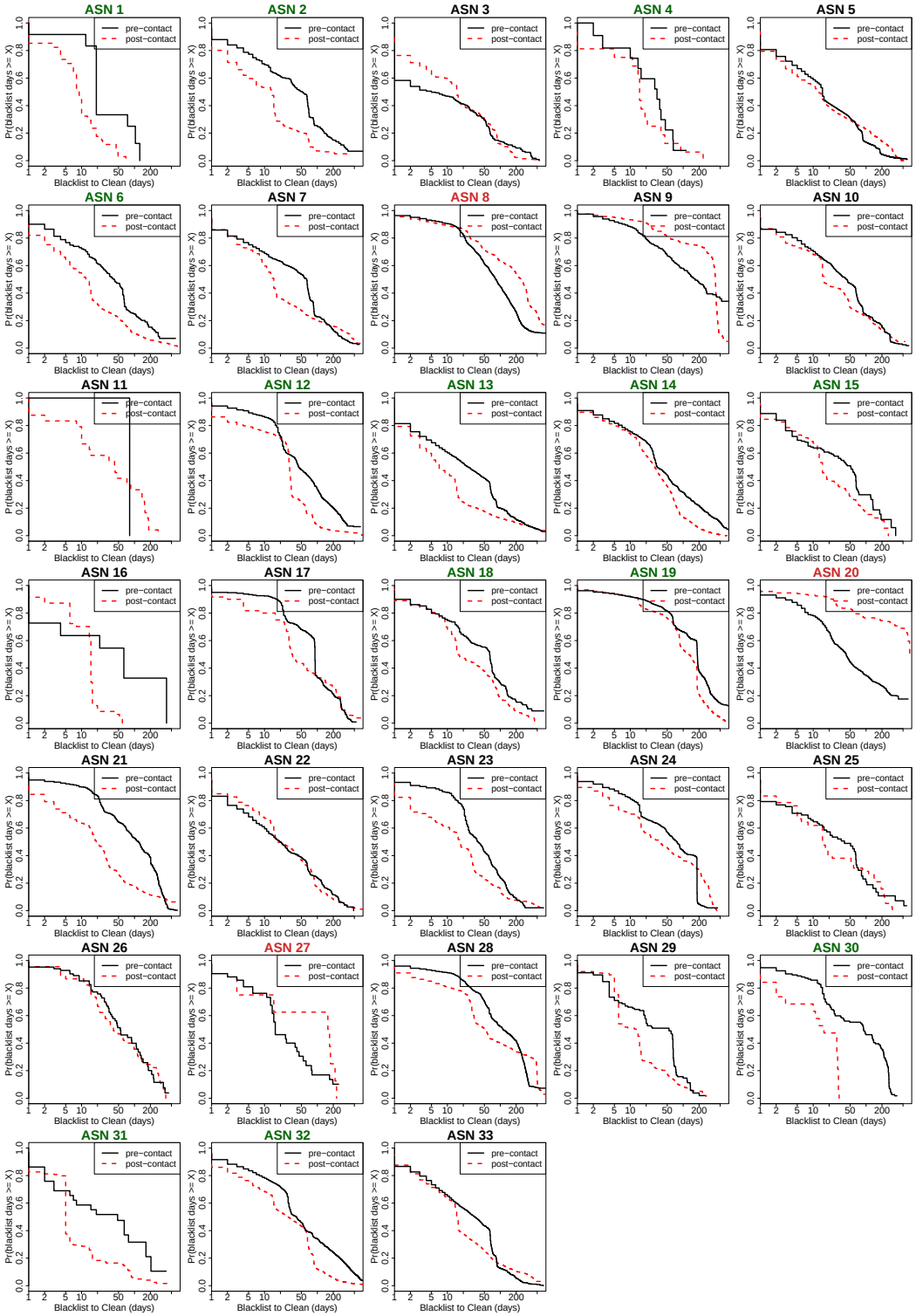

Figure 2.6: Survival probability for the time from blacklist to clean, for the two years before and after contact with StopBadware. Figures are titled green if the AS improved after contact, and red if they worsened.

	#	Δ days to clean	Δ recomp. rate
Improved	13	58	0.010
Worsened	3	-176	0.085
Unclear	17	13	0.008

Table 2.2: Comparing cleanup rate and recompromise rate on the 33 reported-to ASes with 10 or more URLs ever blacklisted on them.

to additionally protect the URL; furthermore, recompromise rates might be outside the reach of the AS (for instance, if their customers use WordPress or other popular content management software). On the other hand, on a statistical level, the shorter time to clean the URL, the longer potential time to recompromise the URL. Furthermore, some ASes might achieve quicker cleanups by cutting corners and not eradicating the root cause of the compromise, e.g., by simply deleting files but not updating software or closing backdoors.

Table 2.2 shows summary statistics for these ASes in aggregate – 13 of these ASes had consistently improved cleanup after receiving a report from StopBadware, whereas 3 ASes worsened their cleanup trend. We labeled providers as *improved* if the survival probability is lower post-contact for more than 85% of the days observed. Similarly, a provider is labeled *worsened* if the survival probability is lower post-contact for fewer than 10% of the days. The progress for all other providers are labeled *unclear*. It is heartening that more ASes seem to improve than worsen, though most ASes do not exhibit a statistically clear trend. Particularly, we notice that the improved ASes have the best improvement in average blacklist to clean time (shortening by two months), whereas the worsened ones get half a year worse.

Furthermore, we found that improved hosting providers generally cleaned the URLs StopBadware reported to them faster. As can be seen in Figure 2.3, of the 13 improving providers, five clearly performed better than average on reported URLs, and none performed clearly worse. Of the three worsening providers, two performed clearly worse than average. Of the remaining 17 providers, for whom data sharing had an unclear long-term effect, six clearly performed better than average and one did worse on cleaning up reported URLs. From this analysis, we conclude that there may be a link between the performance of

providers in cleaning up data from bulk reports and their long-term performance. We explore that possibility in greater detail next.

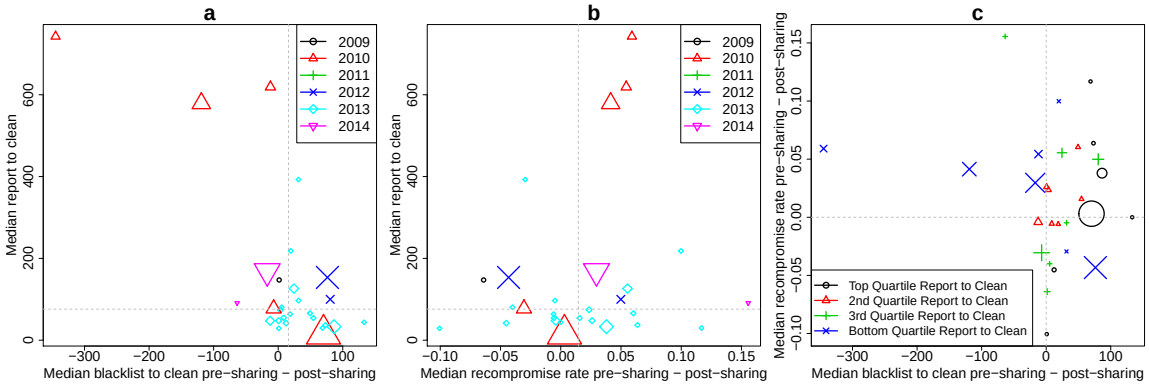

Figure 2.7: Interactions between clean measures: comparing change in the blacklist to clean time, change in the recompromise rate, and the report to clean time for StopBadware reported URLs. Points are scaled by the number of URLs shared (more URLs shared for larger points).

Figure 2.7 examines the interactions between our metrics on how hosting providers clean up malware. We hypothesized that these all would be correlated: that hosting providers who cleaned up the malware we sent them would also improve their cleanup rate and lower their recompromise rate after a report.

Figures 2.7[a] and 2.7[b] explore the relationship between efforts to clean shared URLs and the long-term metrics just presented. The vertical axis in both graphs is the median report to clean time for the shared URLs. The point size is proportional to the number of URLs shared. We change the point icon and color based on the year the report was sent, since trends in sharing (as well as size of malware infections) could change over time. In Figure 2.7[a] the horizontal axis is difference in median blacklist to clean time before and after sharing, where each time period is cut to no more than two years before/after a malware report. In Figure 2.7[b] the horizontal axis is the difference in median recompromise rate before and after sharing. In both cases, positive numbers indicate improvement (i.e., the median blacklist to clean time has gotten shorter).

Figure 2.7[a] exhibits an approximately linear downtrend. This suggests that there is a correlation between the speed of cleanup in response to shared URLs and the improvement (or lack thereof) in cleaning up malware for the period after sharing takes place. By contrast,

Figure 2.7[b] shows no discernible trend. This (discouragingly) shows that the recompromise rate is largely orthogonal to a provider's responsiveness to cleaning URLs that have been shared.

Figure 2.7[c] compares the interaction of both long-term metrics, in hopes of demonstrating whether or not hosting providers improved their malware incident response over the long haul after they received assistance from StopBadware. The horizontal axis plots the change in the median survival blacklist to clean time from the period before to the period after sharing has taken place. Positive numbers indicate improvement (i.e., the median blacklist to clean time has gotten shorter). The vertical axis plots the change in the recompromise rate after sharing. Again, positive numbers indicate improvement. Points are scaled by the number of URLs shared, and they are color-coded according to their relative performance in the report to clean time for shared URLs. Overall, it is striking that most hosting providers improve on at least one of the two measures – only two providers appear in the lower left quadrant, indicated that their performance worsened on both measures. Many providers improved on one measure but not both (top left and bottom right quadrants). The top-performing hosting providers appear in the top right quadrant.

We can see that those providers that responded most quickly to the reports from StopBadware also tended to improve their time to clean long afterwards (most of the top quartile appears in the right quadrants). The top performers over the long term tended to process a smaller number of reports, whereas those processing more reports were more likely to reduce either the recompromise rate or the median blacklist to clean time, but not both.

2.2.3 Matched Pair Analysis

Ideally, we would be able to compare the world where we reported to an AS against the world where we did not in order to isolate the direct effect of reporting bulk URLs to ASes. However, this is obviously not possible. Instead, we attempt to replicate this approach by matching each reported-to AS to a sister AS. These matched pairs have a similar level of compromise on each AS's reporting date and are located in the same country[5]. We assume

[5]In one case, the AS is the only one in the country with malware; here we use the AS with a similar level of malware that is based in the United States instead.

that without our direct intervention, each AS in the pair would have the same compromise level. Thus, significant differences in trends between the reported-to ASes and their sister (matched pair) ASes indicates an effect from reporting.

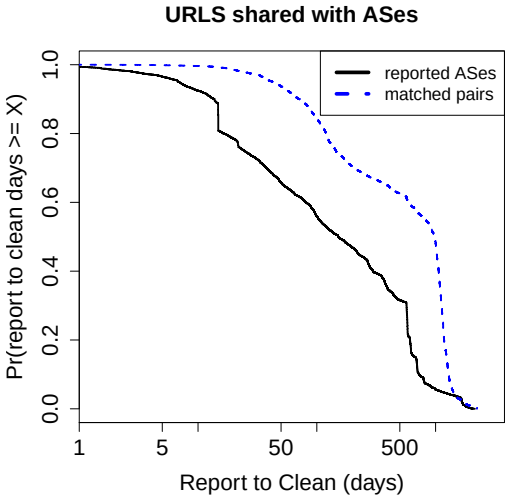

Figure 2.8: Survival probability for the number of days from report to clean for the reported URLs compared to the survival probability for URLs in the matched pair ASes

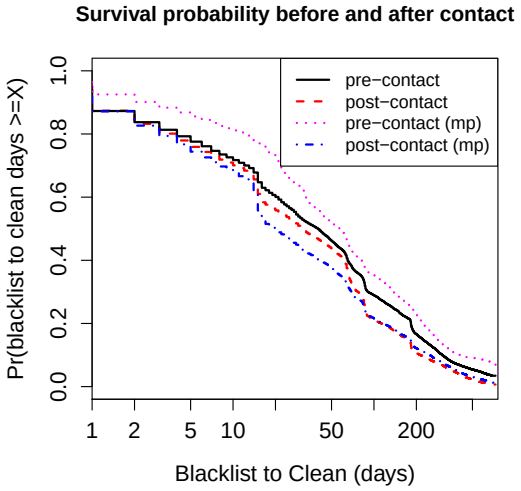

Figure 2.9: Survival probability for the blacklist to clean time for URLs pre- and post-contacting the host (all hosting providers).

First we compare the report to clean time from URLs blacklisted in our reported-to ASes with the URLs blacklisted in their sister ASes on the day we reported to the ASes (Figure 2.8). For the sister ASes (where no report was actually issued), we compute the report to clean time by identifying all URLs that would have been shared had a report been requested and measuring the time to clean starting from the day sharing would have occurred.

We see that URLs that were reported (black line) were cleaned up quicker than URLs that were not (dashed blue line). Half of the reported URLs were cleaned up within 100 days whereas half of the matched pair URLs were cleaned up in over 500 days. Additionally, the recompromise rate for shared URLs is 4% whereas for matched pair URLs have a recompromise rate of 10%. This provides clear evidence that reporting has a large direct effect on improving the cleanup times for the URLs that are shared.

But what about the long-term indirect effects of sharing? Figure 2.9 compares the blacklist to clean time before and after contact for both the reported to ASes (exactly like Figure 2.2) and their sister ASes. We see that the pink dotted line (sister ASes pre-contact) is much higher than the black solid line (reported-to ASes pre-contact). This indicates that before the date StopBadware sent reports to the ASes, their sister ASes took a longer time to clean up a URL. In the period of time after the notification, the reported-to ASes (red dashed line) are about as effective at cleaning malware as their sister ASes (blue dot dashed line).

This analysis suggests that the long-term improvement observed after contact may be caused by some other factor besides receiving malware reports from StopBadware. Nonetheless, individual variation by AS (e.g., among the ASes that significantly improved performance like those appearing in the top right quadrant of Figure 2.7[c] may be affected by sharing data. Further investigation is needed.

2.3 Concluding remarks

The analysis in this chapter helps quantify the impact of sharing abuse data with interested providers. It demonstrates how the responses even from well-intentioned, proactive operators can vary widely. We found that providers cleaned up most of URLs that we directly reported. Even though the cleanup could still take weeks or many months, we are confident that these reports helped the remediation of the shared URLs. However, the evidence that these one-off reports helped improve security over the long term or reduce the prevalence of malware is weak. On the one hand, we should be encouraged that sharing data can clearly improve outcomes for some providers. On the other hand, it is concerning that many providers do not improve after receiving actionable abuse data. Thus, the study at once demonstrates the potential and the limits of sharing security data between private actors.

CHAPTER 3

API FOR SHARING ABUSE DATA WITH HOSTING PROVIDERS

3.1 Introduction

To extend the work of the previous study, we have designed a system to allow hosting providers to sign up for automated access to these bulk reports instead of relying on email requests. In addition to providing this service, the system will be a data collection opportunity to measure the effectiveness of this solution compared to what was discussed in Chapter 2. The requirements below came about from a consultation with StopBadware director Tyler Moore and StopBadware researcher Marie Vasek.

3.2 Requirements

3.2.1 Basic Requirements

Users should be able to access relevant subsets of the StopBadware Clearinghouse data. This access should be programmatic and use the universal transfer protocol, HTTPS. As a basic security requirement, the machine directly serving users on the public Internet should be logically separated from any other University machine, including the StopBadware Clearinghouse data server.

3.2.2 External-facing Requirements

Appropriate HTTP methods and URLs should be used to access the resource, and the server's response should be in the open data-exchange format JSON and should include error messages with HTTP status codes when applicable. Users should not be able to access a blacklist subset more than once each 30 seconds, and no more than 500,000 URLs should be shared at one time, as per StopBadware's data use agreements with it's data providers.

3.2.3 Security Requirements

StopBadware must vet hosting providers before allowing them to register for the service. This process will remain outside the scope of the system described in this thesis; the system must have an authentication mechanism that relies on the authentication achieved from manually vetting.

Also, the system should have reasonable defenses in place to prohibit unauthorized access to the privileged files on the web server or on other parts of the University's infrastructure. This includes restricting SSH access and using an appropriately-configured firewall. Unauthorized attempts to access the Clearinghouse data or any web server resource should be logged, and all communications with users must be encrypted.

3.2.4 Maintenance Requirements

The frameworks and programming languages chosen to build the system should be flexible to accommodate future additions. A version control system must be used during development, and source code should be modular, commented throughout, and written with appropriate variable, function, and file names.

The system must maintain a log for maintenance purposes that includes warning and error entries when bugs or data inconsistencies are encountered. The system will be clearly versioned: the major number is increased for releases that add significant functionality or break compatibility with previous user requests, the minor number is increased for minor functionality additions and major fixes, and the revision number is incremented when minor bugs are fixed.

3.3 Architecture

Figure 3.1 shows the high-level architecture of the implemented system. The system's server is a virtual machine (VM) which communicates with the existing StopBadware data server. The main application code that serves user requests is found in the three Python files `api.py`, `fetching.py`, and `utils.py`. Two other scripts register new users and maintain a relevant subset of StopBadware's Clearinghouse data. This subset includes

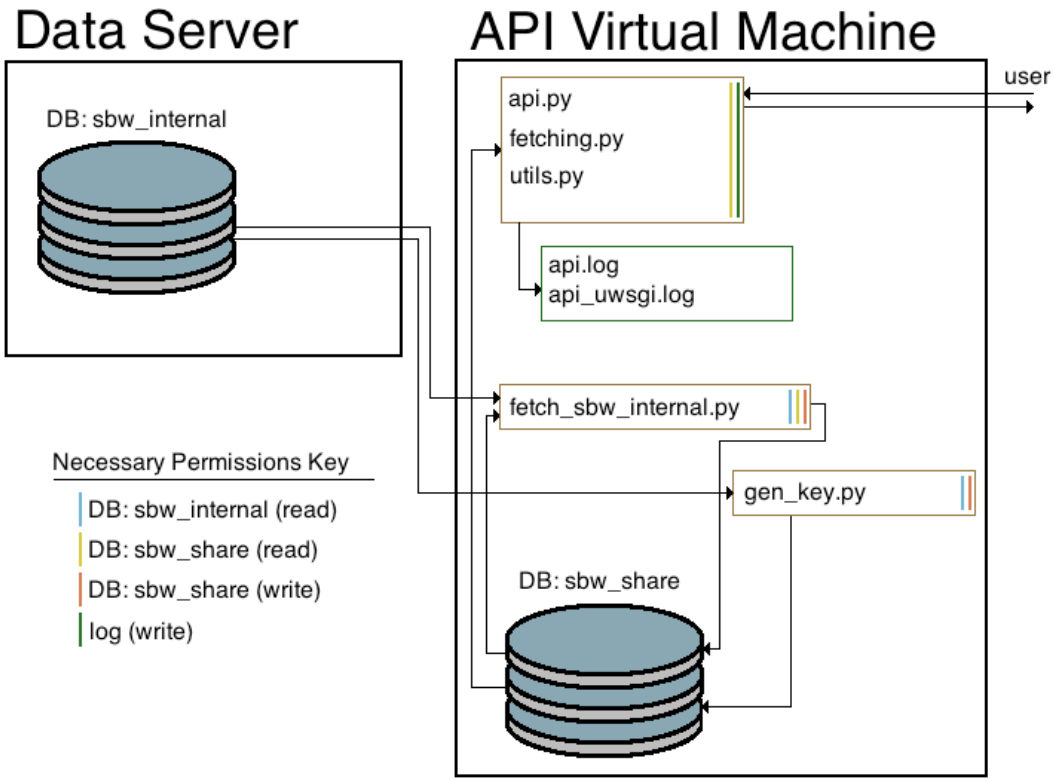

Figure 3.1: Architecture Diagram for StopBadware API

all database records associated with currently-blacklisted URLs that belong to all registered AS owners.

3.3.1 System Architecture Implementation

The system's virtual machine runs the Ubuntu 14.04 operating system and is assigned a public-facing IP address for connecting with users as well as an internally-facing IP address for securely connecting to the data server. Both the data server and the web server use a MySQL relational database, and the websever uses the Python module `PyMySQL` to send queries to and receive responses from `sbw_internal`. The user `"sbw_share_api"@<VM internal IP address>` has been granted read access to `sbw_internal` by the data server's MySQL server.

To serve user requests, an Nginx HTTP server is used with uWSGI as its interface to the application written in Python. The default Nginx configuration was modified to pass only valid SSL traffic to the uWSGI socket. The uWSGI initialization file points to

the main Python application module, `api` and to the Python virtual environment directory used by the application.

To correctly start the web server's MySQL server at startup, its Upstart configuration file was edited so that MySQL will start just before uWSGI starts. This ensures that MySQL is running before uWSGI starts the Python application that connects to the MySQL server.

3.3.2 Registering Users

The script `gen_key.py` can be run by an administrator from a web server terminal to register a new user. The administrator provides the AS number and the script fetches the AS's record from the StopBadware Clearinghouse database then generates a twelve-byte random key. The script saves the key in the local database, `sbw_share`, then the key can be given to the user. The user must supply this key with each subsequent request for identification and authentication. As mentioned before, potential users must be manually vetted before registering.

3.3.3 Database Maintenance and Structure

The script `fetch_sbw_internal.py` is run using the Cron job scheduling tool to make periodic updates to the tables of `sbw_share`. The `sbw_share` database only includes URLs for hosting providers who use the system. It is designed to minimize risk of exposing sensitive URLs from the clearinghouse. The structure of this database is mirrored from `sbw_internal` and includes five relevant tables: `'asns'`, `'ips'`, `'ips_urls'`, `'urls'`, and `'reviews'`. The relevant tables and fields of this database are shown in Table 3.1 through Table 3.5. Each table represents a concrete data object, except for `'ips_urls'`, which is used to associate IP addresses with URLs. Besides these, `sbw_share` has an `'asn_keys'` table (Table 3.6) which stores the randomly-generated keys given to registered users.

3.3.4 Processing User Requests (Software Architecture)

Once user HTTP requests are passed from Nginx to uWSGI, they are processed by the Python application itself. This application uses the Flask web development framework and is organized into three files: `api.py`, `fetching.py`, and `utils.py`.

name	MySQL type	description	alias
`id`	int(11)	unique, incrementing record identifier	`asn_id`
`num`	int(11)	AS number	
`name`	varchar(255)	AS name	
`country_code`	varchar(8)	ISO 3166-1 alpha-2 AS country code	
`created_at`	datetime	date and time this record was created	
`updated_at`	datetime	date and time this record was updated	

Table 3.1: Relevant Table Columns for StopBadware Clearinghouse Data: `'asns'`

name	MySQL type	description	alias
`id`	int(11)	unique, incrementing record identifier	`ip_id`
`num`	int(11)	integer value of IP address	
`asn_id`	int(11)	id for `'asns'` table	
`created_at`	datetime	date and time this record was created	
`updated_at`	datetime	date and time this record was updated	

Table 3.2: Relevant Table Columns for StopBadware Clearinghouse Data: `'ips'`

name	MySQL type	description
`ip_id`	int(11)	id for `'ips'` table
`url_id`	int(11)	id for `'urls'` table
`association_created_at`	datetime	date and time this association was created

Table 3.3: Relevant Table Columns for StopBadware Clearinghouse Data: `'ips_urls'`

name	MySQL type	description	alias
`id`	int(11)	unique, incrementing record identifier	`url_id`
`uri`	varchar(4096)	full URL being blacklisted	
`created_at`	datetime	date and time this record was created	
`updated_at`	datetime	date and time this record was updated	

Table 3.4: Relevant Table Columns for StopBadware Clearinghouse Data: `'urls'`

name	MySQL type	description	alias
`id`	int(11)	unique, incrementing record identifier	
`url_id`	int(11)	id for `'urls'` table	
`status`	varchar(255)	whether this is an active or inactive blacklisting	
`created_at`	datetime	date and time this record was created	
`updated_at`	datetime	date and time this record became inactive	

Table 3.5: Relevant Table Columns for StopBadware Clearinghouse Data: `'reviews'`

name	MySQL type	description	alias
id	int(11)	unique, incrementing record identifier	
asn_id	int(11)	id for 'asns' table	
as_key	varchar(255)	12-byte key in hexadecimal notation	
created_at	datetime	date and time this record was created	
updated_at	datetime	date and time this record was updated	
last_request_at	datetime	date and time this AS key was last used	

Table 3.6: Relevant Table Columns for User Data: 'asn_keys'

api.py is where the Flask app object and the high-level logic serving user requests are defined. First, requests are routed to the appropriate function based on the URL in the HTTP request. Since the API only provides access to a single resource (blacklisted URLs), only one URL path is given a non-error response: /blisted_urls. Requests to the root path are served an HTTP 400 Bad Request message, and requests to all other paths are served HTTP 404 Not Found messages.

Requests to /blisted_urls are passed to the api.request_urls() function. This function checks if the mandatory key parameter was provided in the HTTP request, initializes a connection to the local database, calls a function to get the blacklisted URLs associated with the provided user key, and returns these URLs in a JSON object. If the users did not supply the key parameter, api.request_urls() returns with an HTTP 400 Bad Request message.

The connection to sbw_share is made in api.request_urls() so that a new connection is made with each user request. This allows the application to serve only the most updated data. The result of this connection is a PyMySQL cursor object, which is passed (along with the user's key) to fetching.get_urls_from_key(k, c).

fetching.get_urls_from_key(k, c) contains the mid-level logic needed to get from an AS key to a list of actively-blacklisted URLs belonging to that AS. For this, there are four relations that need to be used: user key to AS number, AS number to the IP addresses belong to that AS, and from these IP addresses to the actively-blacklisted URLs that belong to them. These require queries to sbw_share.'asn_keys', sbw_share.'ips', sbw_share.'ips_urls', and sbw_share.'urls' respectively. Each of these operations is accomplished by a function in fetching.py which are called sequentially by

`fetching.get_urls_from_key(k, c)`, and if no results can be found at any step, `fetching.get_urls_from_key(k, c)` returns a string describing the lack in place of the list of actively-blacklisted URLs.

The functions in `utils.py` accomplish three main tasks: sanitizing input to MySQL queries, enforcing rate limits, and formatting JSON responses. This last task is performed at the end of `api.request_urls()` with a call to `utils.org_data(urls)`, which makes a JSON object including the API version number. If needed, `utils.err_msg(s)` will format error messages that include 204 No Content, 400 Bad Request, 404 Not Found, 429 Too Many Requests, and 500 Internal Server Error.

3.3.5 Logging

Nginx keeps an access log containing each HTTP request and another that logs forbidden requests. uWSGI makes a similar log of each access but additionally includes the number of bytes in the response and the time it took the application to generate the response.

The `logging` Python module is used to log application-level events. Entries at the `INFO` level are lists of `url_ids` associated with each blacklist subset sent to a user. Entries at the `WARNING` level are made when the application fails to update the `last_request_at` field (the only database write operation in the application) or when an inconsistency is found in the database (such as when a query returns multiple records when filtering with a unique column).

3.3.6 Other Security Measures

To secure the web server from attack, Ubuntu's ufw tool was used to only allow incoming traffic on ports 22 (ssh) and 443 (HTTPS). Remote login to the root user was also disabled in the SSH configuration. Also, in the Nginx configuration, the server was set to only provide "nginx" in the server HTTP response header field rather than giving more detailed information about the system.

3.3.7 Future Work

A number of improvements to the API could be added to enhance its functionality. We review several possibilities here.

To mitigate unauthorized access, the system could be made to verify that the source IP address of each request belongs to the AS whose user key is used in the request. Also, due to inconsistencies found in the StopBadware Clearinghouse data, the system should automatically check that the shared URLs do in fact resolve to the user's AS.

The system could also be made to fetch blacklisted URLs for certain IP ranges rather than a whole AS so that smaller hosting providers could sign up for the API.

Beyond these, a caching system should be added before a large number of requests can be served. Also, the number of total blacklisted URLs should be in the JSON response so that users will know how much, if any, of their response was truncated by the rate limit. Finally, these truncated responses could be made to prioritize the fresher data, since StopBadware's data providers send updates at varying frequencies.

CHAPTER 4

CONCLUSION

While a hosting provider's technical staff may or may not be effective at cleaning up compromised sites on their network, we show that notifying them of compromised URLs improves their ability to clean those URLs. The long-term results of reporting blacklisted URLs were varied among hosting providers and no improvement was consistently observed. To improve upon this notification procedure, an API was developed to allow registered hosting providers to programmatically receive lists of the blacklisted URLs hosted on their networks.

4.1 Future Work

Broadly speaking, there are two directions future work could take. First, once the API is deployed it will create an opportunity to collect data on usage. A number of research questions can be investigated. Here is a selection:

- Which hosting providers register for the API and why?
- Does the API yield long-term improvements where one-off emails did not?
- What ways of inviting hosting providers to register for the API prove effective?

Second, future work should focus on identifying ways to improve the overall defensive response to web-based malware. For example, much of the pernicious abuse studied in Chapter 2 and shared with operators were malware URLs that have remained operational for many months or even years. A misalignment of incentives among operators has allowed such malware to stick around far longer than even the criminals would have hoped for. Many current efforts in cleaning networks concentrates on the most recent infections, ignoring such long-standing abuse.

One opportunity for future work inspired by this thesis should be a call to investigate ways to clean up this long-lived abuse. In addition to concentrating efforts on recently compromised URLs, operators should also attempt to clean websites that have been compromised for years. New approaches are needed for these hard cases. Perhaps the dearth of technical information in languages other than English or the lack of resources for non-technical website operators on low cost shared web hosts is finally catching up to us. At the very least, more investigation into why some compromises persist is needed.

We found that sharing bulk data on blacklisted malware URLs with hosting providers was, on average, helpful. Yet bulk malware lists like those that StopBadware shares include lots of long-lived abuse. In order to make bulk data sharing more effective, we need to figure out how to eradicate all compromises, not only the new ones.

BIBLIOGRAPHY

[1] Canali, D., Balzarotti, D., AND Francillon, A. The role of web hosting providers in detecting compromised websites. In: *International World Wide Web Conference*. 2013, 177–188.

[2] Cetin, O., Jhaveri, M. H., Gañán, C., van Eeten, M., AND Moore, T. Understanding the Role of Sender Reputation in Abuse Reporting and Cleanup. In: *Workshop on the Economics of Information Security*. 2015.

[3] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., AND Paxson, V. The matter of Heartbleed. In: *Internet Measurement Conference*. ACM. 2014, 475–488.

[4] Jhaveri, M. H., Cetin, O., Gañán, C., Moore, T., AND Eeten, M. V. Abuse Reporting and the Fight Against Cybercrime. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 68. URL: `http://tylermoore.ens.utulsa.edu/csur17.pdf`.

[5] Kührer, M., Hupperich, T., Rossow, C., AND Holz, T. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In: *Proceedings of the 23rd USENIX Security Symposium*. Aug. 2014.

[6] Li, F., Durumeric, Z., Czyz, J., Karami, M., Bailey, M., McCoy, D., Savage, S., AND Paxson, V. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In: *USENIX Security Symposium*. Aug. 2016.

[7] Li, F., Ho, G., Kuan, E., Niu, Y., Ballard, L., Thomas, K., Bursztein, E., AND Paxson, V. Remedying Web Hijacking: Notification Effectiveness and Webmaster Comprehension. In: *International World Wide Web Conference*. Apr. 2016.

[8] Nappa, A., Rafique, M. Z., AND Caballero, J. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. July 2013.

[9] Provos, N., Mavrommatis, P., Rajab, M. A., AND Monrose, F. All your iFRAMEs point to Us. In: *USENIX Security Symposium*. 2008.

[10] *Spamhaus DNSBL Datafeed*. [Online; accessed 22-March-2017]. URL: https://www.spamhaus.org/datafeed/.

[11] Stock, B., Pellegrino, G., Rossow, C., Johns, M., AND Backes, M. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In: *USENIX Security Symposium*. Aug. 2016.

[12] Tajalizadehkhoob, S., Korczynski, M., Noroozian, A., Ganán, C., AND Eeten, M. van Apples, oranges and hosting providers: Heterogeneity and security in the hosting market. In: *Network Operations and Management Symposium (NOMS)*. IEEE/IFIP. 2016, 289–297. DOI: 10.1109/NOMS.2016.7502824.

[13] Vasek, M., AND Moore, T. Do malware reports expedite cleanup? An experimental study. In: *USENIX Workshop on Cyber Security Experimentation and Test*. Bellevue, WA, Aug. 2012.

[14] Vasek, M., Weeden, M., AND Moore, T. Measuring the Impact of Sharing Abuse Data with Web Hosting Providers. In: *ACM Workshop on Information Sharing and Collaborative Security*. ACM, 2016, 71–80. URL: http://tylermoore.ens.utulsa.edu/wiscs16.pdf.

APPENDIX A

USER MANUAL

The StopBadware API is a RESTful API that provides access to a single resource: lists of StopBadware Clearinghouse URLs that are currently blacklisted and belong to a given autonomous system. Users must register by email with StopBadware to gain an access key for the API.

A.1 Get Blacklisted URLS

The only resource available to users is the list of currently-blacklisted URLs that resolve to the user's AS. This cannot be changed by the user, but may be read using an HTTP POST request with the `key` parameter provided in the request body. This ensures that the user key is does not appear in plain text on the network.

Use the following URL and provide your AS key as a parameter in the request body:

`https://hostname/blisted_urls`

The response is a JSON object with the the list of URLs named `urls` and a floating point version number named `version`. An example response is shown in Figure A.1. If the list of active URLs is too long to comply with our providers' data sharing agreements, you will be sent a random subset of the list.

```
$ curl -X POST --data "key=<user_key>" https://localhost/blisted_urls
{
  "urls": [
    "http://someBlacklistedURL.com/",
        ...
    "http://theLastOne.net/"
  ],
  "version": "1.0"
}
```

Figure A.1: Example API Request and Response

A.2 Error Messages

Table A.1 shows the error codes used by the API and a short description of why you may have received one.

HTTP status code	description
204 No Content	Either no IP addresses or active URLs for the AS
400 Bad Request	An incorrect URL path was used or no key was provided
404 Not Found	An incorrect URL path was requested
429 Too Many Requests	The rate limit (1 req/30 sec) has been exceeded
500 Internal Server Error	An internal error occurred; we will be working to fix this

Table A.1: Error Messages

APPENDIX B

MAINTENANCE MANUAL

This manual is to be used as a reference for those maintaining the StopBadware API system. This includes descriptions of the systems configuration and source code as well as instructions for using the script necessary for user registration.

B.1 Generating ASN Keys

Use the script `gen_key.py` when registering new hosting providers. This script will fetch the `sbw_internal.'asns'` record for the provided AS number, save that record to `sbw_share`, generate a new user key for that ASN, and make an entry in `sbw_share.'asn_keys'`.

Usage: `$ python gen_key.py <AS number>`

B.2 Fetching Blacklist Subsets

The script `fetching_sbw_internal.py` should be run periodically using the Cron job scheduling tool. This script obtains a list of registered ASes from `sbw_share.'asns'`, fetches each blacklist entry for URLs associated with these ASes from `sbw_internal`, then saves these records in `sbw_share.'reviews'`. The script also writes updated records to `sbw_share.'asns'`, `sbw_share.'ips'`, `sbw_share.'ips_urls'`, and `sbw_share.'urls'`. One main feature of this script is that it will break up large MySQL "`INPUT INTO...`" statements into multiple statements based on the `MAX_INSERT_SIZE` global variable.

B.3 System Information and Security Measures

The virtual machine web server runs Ubuntu 14.04 and separates client-facing services and internally-facing requests using one public and one internal network interface card.

Also, the web server uses the `ufw` front end to Ubuntu's `netfilter` firewall. The current status can be seen by running

```
$ sudo ufw status,
```

which will show that incoming connections to ports 22 and 443 (SSH and HTTPS) are allowed and all outgoing connections are allowed. Also, the rootkit checkers `chkrootkit` and `rkhunter` have been installed, the output of which can be seen in the `var/mail/` directory.

B.4 Nginx and uWSGI Configuration

The configuration file for the Nginx server is named `api_nginx.conf` and can be found in the application directory and is symbolically linked to the `/etc/nginx/conf.d` directory. For uWSGI Emperor (the process that spawns workers to server HTTP requests), the configuration is found in `/etc/init/uwsgi.conf`. The file `api_uwsgi.ini`, located in the application directory, points uWSGI workers to the main Python module, `api`, and the Flask app object, `app`.

B.5 Debugging Tips

Modifications to the Python application can be made effective by restarting uWSGI with the following command:

```
$ sudo restart uwsgi
```

uWSGI redirects `Stdout` to the log file `api_uwsgi.log`, so look there for print statement and error stack trace output.

APPENDIX C

SOURCE CODE

C.1 Registration and Database Scripts

C.1.1 gen_key.py

```
# This script generates and stores a secret key to be used by
#   an asn owner to request currently blacklisted urls

import sys
import os
import pymysql
import time
import re

#
# helper functions
#####################################################################

# replace date string with the STR_TO_DATE() MySQL function
def repl_date_StD(s):
    s = re.sub(r"(?=20\d\d-\d\d-\d\d )", "STR_TO_DATE('", s)
    # use a lookbehind to match only the close-parens belonging
    #   to the datetime objects
    s = re.sub(r"(?<=:\d\d:\d\d)", "','%Y-%m-%d %H:%i:%S')", s)
    return s

# replace any "None" values with "NULL" in the string
def repl_nones(s):
    return re.sub(r", None", ", NULL", s)

#
# argument handling
#####################################################################
usage = 'USAGE: python gen_key.py as_number'

# check that sufficient cmd line arguments are provided
if not len(sys.argv) in [2]:
```

```
    print usage
    sys.exit()

# capture and validate AS number
asn = 0
try:
    asn = int(sys.argv[1])
except ValueError:
    print usage
    sys.exit()
if asn >= 4200000000: # max non-private 32-bit ASNs
    print usage
    sys.exit()


#
# connect to databases
#####################################################################

# connect to local db (sbw_share)
conn_share = pymysql.connect(
        db='',
        user='',
        passwd='')
c_share = conn_share.cursor()

# connect to remote db (sbw_internal)
conn_internal = pymysql.connect(
    db='',
    user='',
    host='',
    passwd='')
c_internal = conn_internal.cursor()


#
# check asn
#####################################################################

# check that this asn doesn't already have a key
query = "SELECT as_key FROM asn_keys WHERE num=%s;"
c_share.execute(query % asn)

if len(c_share.fetchall()) != 0:
    sys.exit("The provided AS number already has an associated
        key in the sbw_share.`asn_keys`.")

# fetch asn in `sbw_internal`.`asns` table (and check that it
```

```
#    exists)
query = "SELECT * FROM asns WHERE num=%s;"
# pass parameter as a tuple to avoid injection
c_internal.execute(query, (asn,))

as_row = c_internal.fetchall()[0]
if len(as_row) != 6:
    sys.exit("The provided AS number is not in the sbw_internal.
        asns table.")

asn_id = as_row[0]


#
# insert new 'asns' row
#####################################################################

# construct string of 'asns' row for MySQL statement
asn_s = "("
asn_s += ",".join(map(str, as_row[:2]))
asn_s += ",'" # add single quotes around AS name and country code
asn_s += as_row[2]
asn_s += "','"
asn_s += as_row[3]
asn_s += "',"
asn_s += ",".join(map(str, as_row[4:]))
asn_s += ")"
# replace dates with the MySQL function 'STR_TO_DATE(...)'
asn_s= repl_date_StD(asn_s)
# replace any "None" values with "NULL" in the string
asn_s= repl_nones(asn_s)

query = "INSERT INTO asns (id, num, name, country_code,
    created_at, updated_at) VALUES {0} ON DUPLICATE KEY UPDATE id=
    VALUES(id);"
c_share.execute(query.format(asn_s))
conn_share.commit()


#
# generate key
#####################################################################
new_key = os.urandom(12).encode('hex')


#
# insert into asn_keys db table
#####################################################################
```

```
now = time.strftime('%Y-%m-%d %H:%M:%S')

ins_query = "INSERT INTO asn_keys (asn_id, num, as_key,
    created_at, updated_at) VALUES ({0}, {1}, '{2}', '{3}', '{4}')
    ;"

q = ins_query.format(asn_id, asn, new_key, now, now)
c_share.execute(q)
conn_share.commit()
```

C.1.2 fetch_sbw_internal.py

```
# This script fetches a copy of the `sbw_internal`.`asns`,
#   `ips`, `ips_urls`, `urls`, and `reviews` tables and
#   overwrites the associated tables in `sbw_share` on the
#   local server.

import sys
import pymysql
import re

MAX_INSERT_SIZE = 40000

#
# connect to DBs
##############################################################################

# connect to local db (sbw_share)
def connect_share():
    conn = pymysql.connect(
        db='',
        user='',
        passwd='')

    c = conn.cursor()
    return conn

# connect to remote db (sbw_internal)
def connect_internal():
    conn = pymysql.connect(
        db='',
        user='',
        host='',
        passwd='')

    c = conn.cursor()
    return conn
```

```
#
# helper functions
#####################################################################

# replace single-quoted date string with the STR_TO_DATE()
#   MySQL function
def repl_date_StD(s):
    s = re.sub(r"'(?=20\d\d-\d\d-\d\d )", "STR_TO_DATE('", s)
    # use a lookbehind to match only the close-parens belonging
    #   to the datetime objects
    s = re.sub(r"(?<=:\d\d)'", "','%Y-%m-%d %H:%i:%S')", s)
    return s

# replace any "None" values with "NULL" in the string
def repl_nones(s):
    return re.sub(r", None", ", NULL", s)


#
# ASNs
#####################################################################

# connect to DB
conn_share = connect_share()
c_share = conn_share.cursor()

# get list of asns to fetch for (from sbw_share.`asns`)
q = "SELECT id FROM asns;"
c_share.execute(q)
asn_ids = (i[0] for i in c_share.fetchall())

# construct string of asn ids for MySQL statement
asn_ids_s = "("
asn_ids_s += ", ".join(map(str, asn_ids))
asn_ids_s += ")"

# connect to DB
conn_internal = connect_internal()
c_internal = conn_internal.cursor()

# fetch corresponding rows from sbw_internal.`asns`
q = "SELECT * FROM asns where id in {0};"
c_internal.execute(q.format(asn_ids_s))
asns = [i for i in c_internal.fetchall()]

# make a list of asn lists (to avoid making one MySQL INSERT
```

```
#   statement that is too large)
asns_list = []
done = False
while not done:
    if len(asns) <= MAX_INSERT_SIZE:
        # add the rest of asns as one entry
        asns_list.append(((a[0], a[1], a[2], a[3], str(a[4]), str
            (a[5])) for a in asns))
        done = True
    else:
        # add the first MAX_INSERT_SIZE number of rows of
        #   asns and remove those from asns
        asns_list.append(((a[0], a[1], a[2], a[3], str(a[4]), str
            (a[5])) for a in asns[:MAX_INSERT_SIZE]))
        asns = asns[MAX_INSERT_SIZE:]

# disconnect from sbw_internal db
conn_internal.close()

# construct list of strings of `asns` rows for MySQL statement
asns_s_list = []
for l in asns_list:
    # construct string of `asns` rows for MySQL statement
    s = ",".join(map(str, l))
    # replace single-quoted dates with the MySQL function
    #   'STR_TO_DATE(...)'
    s = repl_date_StD(s)
    # replace any "None" values with "NULL" in the string
    s = repl_nones(s)
    asns_s_list.append(s)

# update sbw_share.`asn` rows with sbw_intnernal.`asns` data
q = "INSERT INTO asns (id, num, name, country_code, created_at,
    updated_at) VALUES {0} ON DUPLICATE KEY UPDATE id=VALUES(id);"
for s in asns_s_list:
    c_share.execute(q.format(s))
    conn_share.commit()

# disconnect from sbw_share db
conn_share.close()


#
# IPs
#####################################################################

# connect to DB
conn_internal = connect_internal()
```

```
c_internal = conn_internal.cursor()

# get sbw_internal.`ips` ips that match share asns
q = "SELECT * FROM ips WHERE asn_id in {0};"
c_internal.execute(q.format(asn_ids_s))
ips = [i for i in c_internal.fetchall()]
ip_ids = [i[0] for i in ips]

# make a list of ip lists (to avoid making one MySQL INSERT
#   statement that is too large)
ips_list = []
done = False
while not done:
    if len(ips) <= MAX_INSERT_SIZE:
        # add the rest of ips as one entry
        ips_list.append(((i[0], i[1], i[2], i[3], str(i[4]), str(
            i[5])) for i in ips))
        done = True
    else:
        # add the first MAX_INSERT_SIZE number of rows of ips
        #   and remove those from ips
        ips_list.append(((i[0], i[1], i[2], i[3], str(i[4]), str(
            i[5])) for i in ips[:MAX_INSERT_SIZE]))
        ips = ips[MAX_INSERT_SIZE:]

# disconnect from sbw_internal db
conn_internal.close()

# construct list of strings of `ips` rows for MySQL statement
ips_s_list = []
for l in ips_list:
    # construct string of `ips` rows for MySQL statement
    s = ",".join(map(str, l))
    # replace single-quoted dates with the MySQL function '
       STR_TO_DATE(...)'
    s = repl_date_StD(s)
    # replace any "None" values with "NULL" in the string
    s = repl_nones(s)
    ips_s_list.append(s)

# connect to DB
conn_share = connect_share()
c_share = conn_share.cursor()

# update sbw_share.`ips` rows with sbw_intnernal.`ips` data
q = "INSERT INTO ips (id, num, cidr, asn_id, created_at,
   updated_at) VALUES {0} ON DUPLICATE KEY UPDATE id=VALUES(id);"
for s in ips_s_list:
```

```
    c_share.execute(q.format(s))
    conn_share.commit()

# disconnect from sbw_share db
conn_share.close()


#
# IPs/URLs
##############################################################################

# construct string of ip_ids for MySQL statement
ip_ids_s = "("
ip_ids_s += ", ".join(map(str, ip_ids))
ip_ids_s += ")"

# connect to DB
conn_internal = connect_internal()
c_internal = conn_internal.cursor()

# get sbw_internal.`ips_urls` rows that match ip_ids
q = "SELECT * FROM ips_urls WHERE ip_id in {0};"
c_internal.execute(q.format(ip_ids_s))
ip_url_rows = [i for i in c_internal.fetchall()]
# capture url_ids for use in later sections
url_ids = (i[1] for i in ip_url_rows)

# make a list of ip_url_row lists (to avoid making one MySQL
#   INSERT statement that is too large)
ip_url_rows_list = []
done = False
while not done:
    if len(ip_url_rows) <= MAX_INSERT_SIZE:
        # add the rest of ip_url_rows as one entry
        ip_url_rows_list.append(((x[0], x[1], str(x[2]), x[3])
            for x in ip_url_rows))
        done = True
    else:
        # add the first MAX_INSERT_SIZE number of rows of
        #   ip_url_rows and remove those from ip_url_rows
        ip_url_rows_list.append(((x[0], x[1], str(x[2]), x[3])
            for x in ip_url_rows[:MAX_INSERT_SIZE]))
        ip_url_rows = ip_url_rows[MAX_INSERT_SIZE:]

# disconnect from sbw_internal db
conn_internal.close()

# construct list of strings of `urls` rows for MySQL statement
```

```
ip_url_rows_s_list = []
for l in ip_url_rows_list:
    # construct string of `ips_urls` rows for MySQL statement
    s = ",".join(map(str, l))
    # replace single-quoted dates with the MySQL function 'STR_TO_DATE(...)'
    s = repl_date_StD(s)
    # replace any "None" values with "NULL" in the string
    s = repl_nones(s)
    ip_url_rows_s_list.append(s)

# connect to DB
conn_share = connect_share()
c_share = conn_share.cursor()

# update sbw_share.`ips` rows with sbw_intnernal.`ips` data
#   NOTE: `ips_urls` is unique on `ips_urls`.`url_id`
q = "INSERT INTO ips_urls (ip_id, url_id, assoc_created_at, current) VALUES {0} ON DUPLICATE KEY UPDATE url_id=VALUES(url_id);"
for s in ip_url_rows_s_list:
    c_share.execute(q.format(s))
    conn_share.commit()

# disconnect from sbw_share db
conn_share.close()


#
# URLs
##########################################################################

# construct string of url_ids for MySQL statement
url_ids_s = "("
url_ids_s += ", ".join(map(str, url_ids))
url_ids_s += ")"

# connect to DB
conn_internal = connect_internal()
c_internal = conn_internal.cursor()

# get sbw_internal.`urls` rows that match url_ids
q = "SELECT * FROM urls WHERE id in {0};"
c_internal.execute(q.format(url_ids_s))
url_rows = [i for i in c_internal.fetchall()]

# make a list of url_row lists (to avoid making one MySQL
#   INSERT statement that is too large)
```

```
url_rows_list = []
done = False
while not done:
    if len(url_rows) <= MAX_INSERT_SIZE:
        # add the rest of url_rows as one entry
        url_rows_list.append(((x[0], x[1], x[2], x[3], x[4], x
            [5], x[6], x[7], x[8], str(x[9]), str(x[10])) for x in
             url_rows))
        done = True
    else:
        # add the first MAX_INSERT_SIZE number of rows of
        #   url_rows and remove those from url_rows
        url_rows_list.append(((x[0], x[1], x[2], x[3], x[4], x
            [5], x[6], x[7], x[8], str(x[9]), str(x[10])) for x in
             url_rows[:MAX_INSERT_SIZE]))
        url_rows = url_rows[MAX_INSERT_SIZE:]

# disconnect from sbw_internal db
conn_internal.close()

# construct list of strings of `urls` rows for MySQL statement
url_rows_s_list = []
for l in url_rows_list:
    # construct string of `urls` rows for MySQL statement
    s = ",".join(map(str, l))
    # replace single-quoted dates with the MySQL function
    #   'STR_TO_DATE(...)'
    s = repl_date_StD(s)
    # replace any "None" values with "NULL" in the string
    s = repl_nones(s)
    url_rows_s_list.append(s)

# connect to DB
conn_share = connect_share()
c_share = conn_share.cursor()

# update sbw_share.`urls` rows with sbw_intnernal.`urls` data
q = "INSERT INTO urls (id, parent_id, uri, host, path, query,
    search_uri, md5, domain_digest, created_at, updated_at) VALUES
    {0} ON DUPLICATE KEY UPDATE id=VALUES(id);"
for s in url_rows_s_list:
    c_share.execute(q.format(s))
    conn_share.commit()

# disconnect from sbw_share db
conn_share.close()
```

```
#
# Reviews (individual blacklist entries)
#####################################################################

# connect to DB
conn_internal = connect_internal()
c_internal = conn_internal.cursor()

# get sbw_internal.`reviews` rows that match url_ids
q = "SELECT * FROM reviews WHERE url_id in {0};"

# use the same url_ids string as before
c_internal.execute(q.format(url_ids_s))
rev_rows = [i for i in c_internal.fetchall()]

# make a list of rev_row lists (to avoid making one MySQL
#   INSERT statement that is too large)
rev_rows_list = []
done = False
while not done:
    if len(rev_rows) <= MAX_INSERT_SIZE:
        # add the rest of rev_rows as one entry
        rev_rows_list.append(((x[0], x[1], x[2], x[3], x[4], str(
            x[5]), str(x[6])) for x in rev_rows))
        done = True
    else:
        # add the first MAX_INSERT_SIZE number of rows of
        #   rev_rows and remove those from rev_rows
        rev_rows_list.append(((x[0], x[1], x[2], x[3], x[4],
            str(x[5]), str(x[6])) for x in
            rev_rows[:MAX_INSERT_SIZE]))
        rev_rows = rev_rows[MAX_INSERT_SIZE:]

# disconnect from sbw_internal db
conn_internal.close()

# construct list of strings of `reviews` rows for MySQL INSERT
#   statement
rev_rows_s_list = []
for l in rev_rows_list:
    # construct string of `reviews` rows for MySQL statement
    s = ",".join(map(str, l))
    # replace single-quoted dates with the MySQL function
    #   'STR_TO_DATE(...)'
    s = repl_date_StD(s)
    # replace any "None" values with "NULL" in the string
    s = repl_nones(s)
    rev_rows_s_list.append(s)
```

```
# connect to DB
conn_share = connect_share()
c_share = conn_share.cursor()

# update sbw_share.`reviews` rows with sbw_intnernal.`reviews`
    data
q = "INSERT INTO reviews (id, partner_id, url_id, status, source,
    created_at, updated_at) VALUES {0} ON DUPLICATE KEY UPDATE id
    =VALUES(id);"
for s in rev_rows_s_list:
     c_share.execute(q.format(s))
     conn_share.commit()

# disconnect from sbw_share db
conn_share.close()
```

C.2 Web Application Scripts

C.2.1 api.py

```
# This script defines the Flask app object, directs HTTP
#   requests to functions, and makes log entries for the
#   StopBadware API; this is the main Python module for
#   the API.

from fetching import get_urls_from_key
import utils
from flask import Flask, request, jsonify
import logging
from logging.handlers import RotatingFileHandler
import pymysql


#####################################################################

VERSION = '1.0'
LOG_PATH = '/var/log/api/api.log'
MAX_URL_LIST_LEN = 500000
MAX_REQUEST_INTERVAL = 30 # in seconds


#####################################################################

app = Flask(__name__)

handler = RotatingFileHandler(LOG_PATH, maxBytes=1000000,
    backupCount=10000)
```

```
formatter = logging.Formatter('%(asctime)s - %(module)s.%(
    funcName)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)
app.logger.addHandler(handler)
app.logger.setLevel(logging.DEBUG)

# root uri
@app.route('/')
def index():
    return utils.message('bad request')

# blisted_urls resource uri
@app.route('/blisted_urls', methods=["POST"])
def request_urls():

    # connect to db so that we get an updated db with each
    #   request
    conn = pymysql.connect(
        db='',
        user='',
        passwd='')

    c = conn.cursor()

    if 'key' in request.form:

        # fetch the asn from the key
        urls = get_urls_from_key(request.form['key'], c)

        # if a string is returned or if there are no urls
        if isinstance(urls, basestring) or urls is None:
            return utils.message(urls)

        # check that this list is not too long
        elif len(urls) > MAX_URL_LIST_LEN:
            # replace list with random subset of the url list
            urls = utils.random_subset(urls)

        # separate the active url_ids for logging and active
        #   urls for sending
        ids, just_urls = zip(*urls)

        # log the url_ids to be sent and update the ASN's
        #   associated last_request_at time
        app.logger.info('urls: ' + ' '.join(ids))
        if not utils.update_last_request_at(request.form['key'],
            conn, c):
```

```
            app.logger.warning('Failed to update last_request_at
                for key {0}.'.format(request.form['key']))

        return utils.org_data(just_urls)

    message = {
        'status': 400,
        'message': 'Bad Request: ' + request.url
    }
    resp = jsonify(message)
    resp.status_code = 400

    return resp

# error handling
@app.errorhandler(404)
def not_found(error):
    message = {
        'status': 404,
        'message': 'Not Found: ' + request.url
    }
    resp = jsonify(message)
    resp.status_code = 404

    return resp

if __name__ == '__main__':
    app.run(host=localhost, debug=False)
```

C.2.2 fetching.py

```
# This file contains helper functions for the sbw url-request
#   api, particularly those associated with db queries

import api
import utils

#################################################################

# overall steps to get a list of urls given an asn_key
def get_urls_from_key(k, c):

    asn = get_asn_from_key(k, c)

    # check exit codes for get_asn_from_key(k, c)
    if asn == 'bad_key':
        return asn
```

```
    if asn == '':
        return 'no_asn'

    # check that this AS has not requested urls within the
    #   api.MAX_REQUEST_INTERVAL
    last_request_at =  get_last_request_at_from_key(k, c)
    if utils.request_interval_too_short(last_request_at):
        return 'rate_limit'

    ip_ids = get_ip_ids_from_asn(asn, c)
    if ip_ids == '':
        return 'no_ips'

    url_ids = get_url_ids_from_ip_ids(ip_ids, c)
    if len(url_ids) < 1:
        return 'no_urls'

    active_urls = get_active_from_url_ids(url_ids, c)
    if len(active_urls) < 1:
        return 'no_urls'
    return active_urls

# this function fetches the AS number from `sbw_share`.
#   `asn_keys` based on the provided key
def get_asn_from_key(k, c):
    key = utils.sanitize_key(k)
    if key != '':
        query = 'SELECT_num_FROM_asn_keys_WHERE_as_key=%s;'
        c.execute(query, (key,))
        res = c.fetchall()
        if len(res) > 0:
            return str(res[0][0])
        else:
            return ''
    else:
        return ''

# this function fetches the last_request_at from `sbw_share`.
#   `asn_keys` based on the provided key
def get_last_request_at_from_key(k, c):
    key = utils.sanitize_key(k)
    if key != '':
        query = 'SELECT_last_request_at_FROM_asn_keys_WHERE_
            as_key=%s;'
        c.execute(query, (key,))
        res = c.fetchall()
        if len(res) > 0:
            return res[0][0]
```

```
        else:
            return ''
    else:
        return ''

# this function returns a list of ip_ids from `sbw_share`.`ips`
#   that are associated with the provided ASN
def get_ip_ids_from_asn(a, c):
        asn = utils.sanitize_asn(a)
        if asn != '':
                query = 'SELECT id FROM asns WHERE num=%s;'
                c.execute(query, (asn,))
                res = c.fetchall()
                if len(res) > 0:
                        query = 'SELECT id FROM ips WHERE asn_id
                           =%s;'
                        c.execute(query, (res[0][0],))
                        res = c.fetchall()
                        ids = []
                        if len(res) == 1:
                                ids.append(str(res[0][0]))
                        else:
                                ids = [str(x[0]) for x in res]
                        return ids
                else:
                        return ''
        return ''

# this function returns a list of url_ids associated with the
#   provided ip ids list
def get_url_ids_from_ip_ids(ip_ids, c):
        ids = utils.sanitize_ids(ip_ids)
        if ids != '':
                url_ids = []
                query = "SELECT url_id FROM ips_urls WHERE ip_id
                   =%s;"
                for i in ids:
                        c.execute(query, (i,))
                        res = c.fetchall()
                        if len(res) > 0:
                                url_ids += [str(x[0]) for x in
                                   res]
                return url_ids
        # there were no ids passed
        return ''

# this function adds ip_nums to ip_ids and returns list of
#   tuples containing one of each (and "None" if value is NULL
```

```
#   in db)
def add_ip_nums_to_ip_ids(ip_ids, c):
    ids = utils.sanitize_ids(ip_ids)
    if ids != '':
        ip_ids_and_nums = []
        query = "SELECT num FROM ips WHERE id=%s;"
        for i in ids:
            c.execute(query, (i,))
            res = c.fetchall()
            if len(res) > 0:
                ip_ids_and_nums.append((i, str(res[0][0])))
        return ip_ids_and_nums
        # there were no ids passed
        return ''

# this function adds `sbw_share`.`urls` uri to url_ids and
#   returns list of tuples containing one of each (and "None"
#   if value is NULL in db)

def add_uri_to_url_ids(url_ids, c):
    ids = utils.sanitize_ids(url_ids)
    if ids != '':
        url_ids_and_uris = []
        query = "SELECT uri FROM urls WHERE id=%s;"
        for i in ids:
            c.execute(query, (i,))
            res = c.fetchall()
            if len(res) > 0:
                url_ids_and_uris.append((i, str(res[0][0])))
        return url_ids_and_uris
        # there were no ids passed
        return ''

# this function returns a list of urls who have an entry in
#   `sbw_share`.`reviews` where status='active'
def get_active_from_url_ids(url_ids, c):
        ids = utils.sanitize_ids(url_ids)
        if ids != '':
                active_url_ids = []
        # query for active `reviews` table rows associated with
        #   each url_id
                query = "SELECT id FROM reviews WHERE url_id=%s
                   AND status='active';"
                for u in ids:
                        c.execute(query, (u,))
                        res = c.fetchall()
                        if len(res) == 1:
                                active_url_ids.append(u)
```

```
                elif len(res) > 1:
                        active_url_ids.append(u)
                        api.app.logger.warning('A url_id
                            query on reviews table
                            returned more than one active
                            review id. Probably on
                            blacklists of at least 2
                            partners.')
        active_urls = []
        query = "SELECT uri FROM urls WHERE id=%s;"
        for u in active_url_ids:
                c.execute(query, (u,))
                res = c.fetchall()
                if len(res) == 1:
                        active_urls.append(res[0][0])
                elif len(res) > 1:
                        api.app.logger.warning('A url_id
                            query on `sbw_share`.`urls`
                            table resulted in more than
                            one url. url_id = {0}'.format(
                            u))
        return zip(active_url_ids, active_urls)

    # there were no ids passed
    return ''
```

C.2.3 utils.py

```
# This script contains helper functions for the SBW api

import api
from flask import jsonify
import time
import datetime
import random

#####################################################################

# key length in number of hex characters (twice the number of
#   bytes)
KEY_LENGTH = 24


#
# misc helper functions
#####################################################################
```

```
# this function returns a message based on an error key phrase
def message(s):

    if s == 'bad request':
        message = {
            'status': 400,
            'message': 'Bad Request'
        }
        resp = jsonify(message)
        resp.status_code = 400
        return resp

    if s == 'bad key' or s == 'no asn':
        message = {
            'status': 429,
            'message': 'Too Many Requests'
        }
        resp = jsonify(message)
        resp.status_code = 429
        return resp

    if s == 'rate limit':
        message = {
            'status': 429,
            'message': 'Too Many Requests'
        }
        resp = jsonify(message)
        resp.status_code = 429
        return resp

    if s == 'no ips':
        message = {
            'status': 204,
            'message': 'No Content'
        }
        resp = jsonify(message)
        resp.status_code = 204
        return resp

    if s == 'no urls':
        message = {
            'status': 204,
            'message': 'No Content'
        }
        resp = jsonify(message)
        resp.status_code = 204
        return resp
```

```
    if s == 'too_many_urls': # not currently used
        message = {
            'status': 413,
            'message': 'Payload_Too_Large'
        }
        resp = jsonify(message)
        resp.status_code = 204

    message = {
        'status': '500',
        'message': "Internal_Server_Error."
    }
    resp = jsonify(message)
    resp.status_code = 500
    return resp

# this function organizes a list of urls into a dictionary
def org_data(urls):
    # json dates: 2007-11-06T16:34:41.000Z
    data = {
        'version':api.VERSION,
        'date':time.strftime("%Y-%m-%dT%H:%M:%s"),
        'urls':urls
    }
    return jsonify(data)

# this function returns a random subset of the provided list of
#   urls with length of MAX_URL_LIST_LEN
def random_subset(urls):
    indices = (random.randint(0, len(urls)-1) for i in \
        range(0, api.MAX_URL_LIST_LEN))
    subset = [urls[i] for i in indices]
    return subset

# this function updates the appropriate `sbw_share`.`asn_key`
#   last_request_at value to the current time
def update_last_request_at(key, conn, c):
    key = sanitize_key(key)
    if key == '':
        return False
    else:
        now = time.strftime("%Y-%m-%d_%H:%M:%S")
        query = "UPDATE_asn_keys_SET_last_request_at='{0}'_WHERE_
            as_key='{1}';"
        c.execute(query.format(now, key))
        conn.commit()
        return True
    return False
```

```
# returns true if the current time is less than
#   MAX_REQUEST_INTERVAL seconds after the provided time of the
#   last request
def request_interval_too_short(last_request_at):
    now = datetime.datetime.now()
    if last_request_at + datetime.timedelta(0, api.
       MAX_REQUEST_INTERVAL) > now:
        return True
    return False

#
# input sanitation functions
#####################################################################

# check that the key (stripped of leading and trailing
#   whitespace) has only hex numbers and letters
def sanitize_key(k):
    # check to make sure the key is made up of only valid
    #   characters
    key = k.strip()
    if set(k) <= set('0123456789abcdef') and len(k) == KEY_LENGTH
       :
        return k
    else:
        return ''

# check if asn is a number of an apropriate length
def sanitize_asn(a):
    asn = a.strip()
    if set(asn) <= set('0123456789'):
        if len(asn) > 0 and len(asn) < 11:
            return asn
    return ''

# check if list of ids contains only numbers
def sanitize_ip_ids(ids):
    if len(ids) > 0:
        if len(ids) > 1:
            for i in ids:
                if not set(i) <= set('0123456789'):
                    return ''
        else:
            if not set(ids[0]) <= set('0123456789'):
                    return ''
    return ids

# check if list of ids contains only numbers
```

```
def sanitize_ids(ids):
    if len(ids) > 0:
        if len(ids) > 1:
            for i in ids:
                if not set(i) <= set('0123456789'):
                    return ''
        else:
            if not set(ids[0]) <= set('0123456789'):
                    return ''
    return ids
```