HASCHK: Malicious Download Mitigation Leveraging High Availability Systems

Anonymous Author(s)

Abstract

Downloading resources over the internet comes with many risks, including the chance that an attacker has replaced your desired resource with a compromised version. The de facto standard for addressing this risk is the use of *checksums* coupled with a secure transport layer; users download a resource, compute its checksum, and compare that with an authoritative checksum. Problems with this include (1) *user apathy*—for most users, calculating and validating the checksum is too tedious; and (2) *co-hosting*—an attacker who compromises a resource can trivially compromise a checksum hosted on the same system. In this paper we propose *HASCHK*, a novel resource validation approach meant as a complete replacement for current checksum-based approaches. HASCHK implementations automate the tedious parts of checksum verification to sidestep user apathy while leveraging authenticated highly available distributed systems to address co-hosting. We carefully evaluate the security, performance, and practicality of our approach through proof-of-concept DNS and DHT based implementations in Google Chrome; implementations are tested versus common resource integrity violations. While not a panacea, we find that our approach is more effective than existing mitigation methods, significantly raises the bar for the attacker, and is deployable at scale.

1 Introduction

In 2010, through compromising legitimate applications available on trusted vendor websites, nation-state actors launched the Havex malware, targeting aviation, defense, pharmaceutical, and other companies in Europe and the United States [28, 39]. In 2012, attackers compromised an official phpMyAdmin download mirror hosted by reputable software provider SourceForge. The backdoored version of the popular database frontend was downloaded by hundreds of users, potentially allowing attackers to gain access to private customer data [43, 44]. In 2016, attackers broke into the Linux Mint distribution server and replaced a legitimate Mint ISO with one containing a backdoor, infecting hundreds of machines with the malware [4, 49]. Over a four day period in 2017, users of the popular HandBrake open source video transcoder on Mac/OSX were made aware that, along with their expected video software, they may have also downloaded a trojan that was uploading their sensitive data to a remote server [27]. HandBrake developers recommended users perform checksum validation to determine if their install was compromised [26].

As every internet user is certainly aware, downloading resources over the internet comes with considerable risk. This risk can be divided into three broad concerns: response authentication, communication confidentiality, and resource integrity. Response authentication allows us to determine if a response received indeed originates from its purported source through, for instance, the adoption of a Public Key Infrastructure (PKI) scheme [11]. Communication confidentiality, on the other hand, allows us to keep the data transacted between two or more parties private except to said parties through some form of encryption, such as AES [14]. Finally, resource integrity allows us to verify that the data we are receiving is the data we are expecting to receive.

When it comes to response authentication and communication confidentiality concerns on the internet, the state of the art in attack mitigation is Transport Layer Security (TLS) and its Hyper Text Transfer Protocol (HTTP)/PKI based implementation, HTTPS [8, 11, 17, 18, 47]. Assuming well behaved certificate authorities and modern browsing software, TLS and related protocols, when properly deployed, mitigate myriad attacks on authentication confidentiality.

However, as a *communication* protocol, TLS only guarantees the integrity of each *end to end communication* via message authentication code (MAC) [18]. But protected encrypted communications mean nothing if the contents of those communications are corrupted before the fact. Hence, the integrity of resources at the application layer (rather than the transport layer) is outside of the model addressed by TLS and HTTPS [18, 47].

Attacks on resource integrity can be considered a subset of *Supply Chain Attacks* (SCA). Rather than attack an entity directly, SCAs are the compromise of an entity's software source code (or any product) via cyber attack, insider threat, upstream asset compromise, trusted hardware/vendor compromise, or other attack on one or more phases of the software development life cycle [42]. These attacks are hard to detect, even harder to prevent, and have the goal of infecting and exploiting targets and victims by abusing the trust between consumer and reputable software vendor [40].

Ensuring the integrity of resources exchanged over the internet despite SCAs and other active attacks is a hard and well studied problem [1, 2, 9, 11, 22, 35, 38, 47]. For a long time, the de facto standard for addressing this risk in the generic case is with the use of *checksums* coupled with some secure transport medium like TLS/HTTPS. Checksums in this context are cryptographic digests generated by a cryptographic hashing function run over the resource's file contents. When a user downloads a file from some source, they are expected to run the same cryptographic hashing function over their version of the resource, yield a local checksum, and match it with the authoritative checksum given to them from said source.

However, checksums come up short as a solution to the resource integrity problem. Foremost is a well-understood but harsh reality: *user-apathy*—a non-trivial number of users will not be burdened with manually calculating checksums for the resources they download. While detailing how they gained unauthorized access to the servers, one of the hackers behind the 2016 breach of Linux Mint's distribution system went so far as to comment (in respect to checksums): "Who the [expletive] checks those anyway?" [51]. Hardly unique to checksum calculation, cryptographic schemes from HTTPS to PGP have found user apathy a difficult problem space to navigate [3, 52].

Even if a user felt the urge to manually calculate a checksum, they must search for the corresponding authoritative checksum to verify that calculation. As there is no standard storage or retrieval methods for checksums, they could be stored anywhere, or even in multiple different locations that must then be kept consistent; users are not guaranteed to find an authoritative checksum, even if they are published online somewhere. If they do manage to find the authoritative checksum and also recognize the checksums are different, the user is then expected to "do the right thing," whatever that happens to be in context.

Then there is the futility of *co-hosting* a resource and its checksum on the same distribution system. While cost-effective compared to hosting two or more discrete systems—one for the resource and one for the resource's checksum—an attacker that compromises a single distribution system hosting a resource and its checksum can mutate both, rendering the checksum irrelevant. The co-hosting problem was demonstrated by the 2016 hack of Linux Mint's distribution server [4, 49].

Checksums as they are employed currently are not effective at guaranteeing resource integrity. Recognizing this, some corporations and large entities rely instead on PKI-based approaches such as digital signature validation and code signing [11]. These roll-your-own solutions, often proprietary, have been deployed successfully to mitigate resource integrity attacks in mature software package ecosystems (*e.g.,* Debian/apt, Red Hat/yum, Arch/pacman) and walled-garden app stores like Google Play, Apple App Store, and the Microsoft Store.

Unfortunately, not all resources available on the internet are acquired through software package ecosystems with built-in PKI support, nor are all resources software binaries; moreover, these PKI schemes are not compatible with one another and cannot scale to secure arbitrary resources on the internet without significant cost and effort. Worse, roll-your-own PKI is hard to get right [11], implying systems built atop them are inherently more *fragile*—susceptible to malfunction due to small errors or misconfigurations.

In this paper, we propose HASCHK, a novel approach for verifying the integrity of resources downloaded over the internet that is a complete replacement for traditional checksums.

We view the problem with four key concerns in mind: a) implementations must provide security guarantees transparently without adding any extra burden on the end user in the average case—here, an optimal solution avoids relying on the user to overcome **user apathy** in the interest of security; b) configuring the validation method is simple for operators and developers to integrate and deploy while ensuring configuration of a potentially-expensive discrete secondary system to host checksums (*i.e.,* **co-hosting**) is unnecessary; c) the validation method is not tightly coupled with any particular highly-available system; and d) no application or website source code changes or end-user facing web server/web infrastructure alterations are necessary.

Of these concerns, the first two are paramount.

We implemented HASCHK as two proof-of-concept Google Chrome extensions, one relying on DNSSEC-secured DNS as a highly available backend while the other relies on an OpenDHT-based dummy backend.

We then evaluate the security, scalability, and performance of our automated defense against resource corruption and demonstrate the effectiveness and practicality of the HASCHK approach. Specifically, we find no additional obstacles to efficient deployment at scale outside of those imposed by the chosen authenticated distributed high availability system.

For our proof-of-concept implementations, we provide a publicly accessible empirical demonstration of HASCHK's protective utility via a patched HotCRP instance (cf. Section 7). We observed no download performance overhead compared to downloads without HASCHK during our evaluation.

In summary, our primary contributions are:

- We propose a practical approach to defending against receiving corrupted or compromised resources over the internet. Contrasted with current solutions, our concrete implementations require no app/website source code or end user facing web server/web infrastructure changes, do not employ unreliable heuristics, do not interfere with other protocols or software extensions that might also deal with download security, and can be transparently deployed without adding to application/infrastructure fragility; *e.g.,* HASCHK-aware clients with the extension will have their downloads secured while the user experience of clients unaware of HASCHK remain completely unaffected.

- We present our prototype HASCHK implementations: DNSCHK and DHTCHK. Both are implemented as Google Chrome extensions. We demonstrate their effectiveness in automatically and transparently mitigating the accidental consumption of compromised resources from a compromised test servers. To the best of our knowledge, this is the first system providing such capabilities with marginal deployment cost and, unlike a traditional checksum-based approach, does not require the end user to overcome apathy in the average case.
- We extensively evaluate the security, performance (overhead), and deployment costs of DNSCHK and DHTCHK. We find that our approach is more effective than checksums for mitigating resource integrity attacks. Further, we show that our implementations are capable of detecting a wide variety of real-world integrity errors, significantly raising the bar for the attacker. Finally, we observed no download performance overhead compared to downloads without DNSCHK or DHTCHK. DNSCHK specifically, as it is backed by the (in)famous global DNS network, is able to be deployable at scale for entities that already secure their DNS zone(s) with DNSSEC.

We release our DNSCHK and DHTCHK proof-of-concept implementations to the community as open source software to promote exploration of the HASCHK approach (cf. Section 7).

2 Background

In this section, we describe the motivation for HASCHK, including four case studies that frame the threat we model against. We then examine current methods to detect and prevent resource corruption including checksums, HTTPS, anti-malware, purely PKI-based solutions, and others.

2.1 Supply Chain Attacks on Resource Integrity

Modern software development requires a complex globally distributed supply chain and development ecosystem for organizations and other entities to design, develop, deploy, and maintain products efficiently [40]. Such a globally distributed ecosystem necessitates integration with potentially many third party entities, be they specialty driver manufacturers, external content distribution network (CDN) providers, third party database management software, download mirrors, etc.

Critically, reliance on third parties, while often cost-effective and feature-rich, also increases the risk of a security compromise at some point in the supply chain [40]. These types of compromises are known as Supply Chain Attacks (SCA). In the context of software development, SCAs are the compromise of an entity's software source code via cyber attack, insider threat, upstream asset compromise, trusted hardware/vendor compromise, or some other attack on one or more phases of the software development life cycle or "supply chain" to infect an unsuspecting end user [42].

Every year, major SCAs become more frequent and their fallout more widely felt [40, 42]. Whether major or minor, SCAs are hard to detect, even harder to prevent, and have the goal of infecting and exploiting victims by violating the trust between consumer and reputable software vendor.

Table 1 details the phases of a generic software development supply chain. For the purposes of this research, we focus exclusively on SCAs targeting the deployment, maintenance, and retirement phases.

2.2 Motivation: Cases

Here, we select four historic attacks we believe most effectively articulate the threat posed by resource integrity SCAs and how HASCHK might have been used to more effectively mitigate fallout. We examine each attack, noting the critical points of failure in their checksum-based resource security model.

Case 1: PhpMyAdmin. For an unspecified amount of time circa 2012, a compromised download mirror in SourceForge's official HTTPS-protected CDN was distributing a malicious version of the popular database administration software phpMyAdmin [13]. The administrator of the mirror in question confirmed the attack was due to a vulnerability not shared by SourceForge's other mirrors [43].

Attackers mutated the software image, injecting files that would allow any attacker aware of their existence to remotely execute arbitrary PHP code on the victim's system [44]. SourceForge estimates approximately 400 unique users downloaded this corrupted version of phpMyAdmin before the mirror was disconnected from their CDN, potentially allowing attackers access to the private customer data of any number of organizations [43].

While the attackers were able to penetrate a mirror in SourceForge's CDN, the official phpMyAdmin website was entirely unaffected; the authoritative checksums listed on the site's download page were similarly unaffected [43]. Hence, a user who was sufficiently motivated, had sufficient

Concept	Design	Development	Integration	Deployment	Maintenance	Retirement
X	X	X	X	✓	✓	✓

Table 1: The software development supply chain. Attacks outside of the deployment, maintenance, and retirement phases are outside of the HASCHK threat model; hence, they are not considered.

technical knowledge of checksums and how to calculate them, and was also privy to the location of the correct checksum for the official phpMyAdmin image *might* have noticed the discrepancy between the two digests. Clearly, a non-trivial number of users do not meet these criteria, and this attack demonstrates the problem of *user-apathy*.

Case 2: Linux Mint. In 2016, the Linux Mint team discovered an intrusion into their official HTTPS-protected distribution server [49]. Attackers mutated download links originally pointing to the Linux Mint 17.3 Cinnamon edition ISO, redirecting unsuspecting users to a disparate system hosting a custom Mint ISO compiled with the IRC-based Linux backdoor malware *Tsunami* [4]. The attack affected hundreds of the downloads during that day, with the attackers claiming that a “few hundred” Linux Mint installs were explicitly under their control. The primary motivation behind the intrusion was the construction of a botnet [51]. The authoritative checksum displayed on the official website was also mutated to corroborate the backdoored ISO [51], illustrating the *co-hosting* problem.

Storing the checksum elsewhere may have prevented mutations on the checksum; still, as demonstrated by the first case, such an effort is not itself a solution. Hosting a checksum on a secondary system is not very useful if users downloading the resource protected by that checksum cannot find it or are not actually *checking* it against a manual calculation.

Case 3: Havex. As part of a widespread espionage campaign beginning in 2010, Russian Intelligence Services targeted the industrial control systems of numerous aviation, national defense, critical infrastructure, pharmaceutical, petrochemical, and other companies and organizations with the Havex remote access trojan [28, 39]. The attack was carried out in phases whereby innocuous software images hosted on disparate *legitimate* vendor websites were targeted for replacement with versions infected with the Havex malware [39]. The goal here, as is the case with all SCAs, was to infect victims indirectly by having the Havex malware bundled into opaque software dependencies, *i.e.,* a hardware driver or internal communication application.

It is estimated that Havex successfully impacted hundreds or even thousands of corporations and organizations—mostly in United States and Europe [39]. The motivation behind the Havex malware was intelligence exfiltration and espionage [28]. How many of these vendors employed checksums and other mitigations as part of their software release cycle is not well reported, though investigators note said vendors’ distribution mechanisms were insecure [39]; however, an automated resource validation method could have helped mitigate the delivery of compromised software to end users.

Case 4: HandBrake. In May of 2017, users of HandBrake, a popular open source video transcoder for Mac/OSX, were made aware that they may have downloaded and installed a trojan riding atop their transcoding software. Attackers breached an HTTPS-protected HandBrake download mirror, replacing the legitimate software with a version containing a novel variant of the *Proton* malware [27]. The number of users potentially affected is unreported.

The goal of the attack was the exfiltration of victims’ sensitive data, including entire keychains (unlocked), private keys, browser password databases, 1Password/Lastpass vaults, decrypted files, and victims’ personal videos and other media [27]. The HandBrake developers recommended users perform manual checksum validation to determine if their installation media was compromised [26].

Despite the attackers mutating the HandBrake binary, the authoritative checksums listed on the official HandBrake download page were reportedly left untouched [26]. Further, the developers of HandBrake store their authoritative checksums both on their official website and in their official GitHub repository [26]. A sufficiently knowledgeable, sufficiently motivated user *might* have noticed the discrepancy between their calculated checksum and the authoritative checksum listed on the download page.

Suppose, however, that the attackers *had* managed to mutate the checksums on the official website. Then there would be a discrepancy between the authoritative checksums on the official site and the authoritative checksums in the GitHub repository—that is *if* users are even aware that a second set of checksums are available at all. On top of technical knowledge, a user in this confusing situation is then expected to “do the right thing,” whatever that happens to be in this context.

2.3 Other Detection and Mitigation Methods

Detecting and/or mitigating resource integrity SCAs and other active attacks is a non-trivial and well studied problem [1, 2, 9, 11, 22, 35, 38, 47]. What follows is a brief overview of current and popular methods to ensure the integrity of resources exchanged over the internet other than checksums and HTTPS/TLS.

2.3.1 Anti-Malware Software

Anti-malware software are heuristic-based programs designed for the specific purpose of detecting and removing various kinds of malware. However, updates to anti-malware definitions often lag behind or occur in response to the release of crippling malware. For example, during the 2017 compromise of the HandBrake distribution mirror, users who first ran the compromised HandBrake image through *VirusTotal*—a web service that will run a resource through several dozen popular anti-malware products—received a report claiming no infections were detected, despite the empirically verifiable presence of the Proton malware.

Worse, not all resource compromises end up looking like malware. In the 2012 compromise of SourceForge's CDN, where a malicious version of phpMyAdmin was delivered to hundreds of users, the PHP source itself was altered to enable remote code execution. However, the extraneous code was virtually indistinguishable from the rest of the raw PHP source in the phpmyAdmin image being distributed.

At the time of writing (2018), VirusTotal correctly identifies the compromised version of the Handbrake image as malware.

2.3.2 Browser-based Heuristics and Blacklists

Modern browsers employ heuristic and blacklist based detection and prevention schemes in an attempt to protect users from encountering malicious content on the internet. Implementations include Google Chrome's *Safe Browsing* feature, Mozilla Firefox's *Phishing/Download Protection*, and Microsoft Edge's *malware sniffing* Windows Defender browser bundle.

Similar to anti-malware software, browser-based heuristics and blacklists are a reactive rather than proactive solution; hence, they are ineffective at shielding users from attacks on the integrity of the resources downloaded over the internet.

3 The HASCHK Approach

In this section we detail the HASCHK approach: a practical defense against receiving corrupted or compromised resources over the internet. We further present our proof-of-concept Google Chrome extension implementations, DNSCHK and DHTCHK.

3.1 Defeating User Apathy

Human factors such as user apathy have stymied cryptographers for decades. Schemes that are otherwise reasonably cryptographically solid can fail catastrophically due to human error, confusion, or simple lack of interest. Some users are likely to avoid using a security measure altogether if it presents even a minor obstacle to immediate gratification [3, 52]. In the browser, for example, this phenomenon can be observed empirically.

Leveraging the in-browser telemetry of Mozilla Firefox and Google Chrome to passively observe over 25 million warning impressions in 2013 (see Fig. **??**), Akhawe et al. found that users of Google Chrome clicked through a *quarter of malware and phishing warnings* and *70% of TLS warnings* [3]. Users also clicked through a third of Mozilla Firefox's TLS warnings and a tenth of their malware and phishing warnings. That is to say: a significant percentage of browser users are *determined* not to let TLS trust issues and/or the threat of malware prevent them from receiving their desired content. Hence, we must assume: some non-trivial number of users, similarly determined to transact resources over the internet, will not be burdened with the off-path minutiae of manually calculating a checksum (if they are even familiar with the jargon) and verifying the integrity of the resources they are downloading.

With this assumption in mind, the primary goal of HASCHK then is to side-step the human factor altogether by providing a completely transparent and unobtrusive in the average case, fully-automated method of checksum calculation and verification in the average case that requires no changes to application logic or source code. We achieve this through 1) the unique identification of individual hosted resources and 2) a highly available mapping of unique resource identifiers to corresponding checksums.

Fig. **??** illustrates the HASCHK approach. HASCHK implementations can be imagined as a security layer sitting between the user and the resource. Immediately after a resource is downloaded, two cryptographic digests are generated. One digest uniquely fingerprints said resource based on its name. This is known as the *Non-Authoritative Checksum* (NAC) and is yielded from running the cryptographic hashing function over the contents of the resource file. The second digest uniquely fingerprints said resource based on its contents. This is known as the *Resource Identifier* (RI) and is yielded from running the cryptographic hashing function over the resource's public path on the distribution system.

Next, HASCHK uses the RI to retrieve an *Authoritative Checksum* (AC) from the backend. If successful, HASCHK will compare the NAC to the AC—we refer to this as *Non-Authoritative Checksum Validation* (NAC Validation). In the case where NAC Validation fails, *i.e.,* they do not match, some implementation-specific action should be taken to mitigate as much as possible the risk to the end user. In most contexts, this means deleting or renaming the unsafe resource, forcing the user to deal with the problem. Otherwise, HASCHK remains completely transparent the the end user, as demonstrated by our browser-based implementations.

3.2 Defeating Co-Hosting

Funding and maintaining a single server/system to host all of your assets can be extremely cost-effective in the short term compared to hosting two or more discrete systems—one hosting the resource and one hosting the resource's checksum. Unfortunately, this establishes a single point of failure: an attacker that compromises such a system can both mutate the resource and update the checksum to match the mutation. Hence, *co-hosting* a resource and its corresponding checksum on the same distribution system virtually negates the effectiveness of having a checksum at all. This is widely understood in the security community [4].

Hence, deployment of HASCHK necessitates the existence of a separate distribution mechanism for resources and for ACs. Though the concept of using some distributed authenticated storage service to query a global mapping between RIs and ACs is not new and seems straightforward, but the problem is more complex than perhaps first meets the eye.

The state of the art in modern fully authenticated highly available schemes are based on some form of PKI; an entity must roll its own PKI solution, actively maintain it, and hope it is bug free. Examples include the Windows Store, Google Play Store for Android, and the Apple Store for iPhone.

There are several problems with purely PKI-based resource integrity verification. For one: a roll-your-own PKI-based model clearly cannot scale to secure *arbitrary* resources on the entire internet. This is doubly true when considering the primary consumers of those resources—end users—likely cannot distinguish a digital signature from, for instance, a key [52]. Further, unlike a standardized approach like HTTPS/TLS or DNSSEC, which are notoriously hard to implement correctly in their own right, rolling your own PKI-based solution is a path fraught with even greater peril. It is well known in the community that these often very complex PKI systems are hard to design correctly, hard implement correctly, and hard to effectively maintain.

Fortunately, there has been a lot of effort put into researching, designing, and standardizing several high availability fully authenticated globally distributed high performance storage technologies, some of which web-facing entities and IT teams are already quite familiar with and most already pay for, *e.g.,* the Domain Name System (DNS). Adding extra resource records to a DNS zone, for instance, is essentially a costless operation, meaning any entity that already has a DNSSEC-protected web presence can immediately deploy HASCHK. This is key to the motivation behind the HASCHK approach as well as the design of our DNSCHK proof-of-concept implementation.

Other candidate high availability systems include DHTs, storage clusters, relational and non-relational databases, and any high availability authenticated key-value store.

3.3 Platform Diversity

From our evaluation (cf. Section 4), the computational overhead of running HASCHK is minimal for most resources. Further, additional network load is negligible. Hence, the HASCHK approach can be incorporated into software on most any device capable of communicating with the chosen backend. This includes desktops, laptops, tablets, mobile devices, embedded systems, etc.

3.4 Proof-of-Concept Implementations

3.4.1 DNSCHK and DHTCHK

We implement HASCHK as two proof-of-concept Google Chrome extensions: DNSCHK and DHTCHK. They work with DNS and Ring OpenDHT as their highly available backends, respectively. Our Chrome extensions do not make any modifications to the Chrome user interface or viewport other than the extension icon in the toolbar. Further, downloads work exactly the same whether or not DNSCHK or DHTCHK are installed; the extensions are transparent to end users. However, if a failure is experienced during NAC Validation (*i.e.,* a "non-average" case), the extensions will alert the user to the dangerous download via toolbar icon and popup interface.

Immediately after a resource download is first detected, the extensions compute an RI from the full URL path of the resource. For the purposes of our proof-of-concept implementations, we calculate the RI as a hash digest of the path component of the URL pointing to the resource. For example, consider a web resource hosted at `https://somesite.com/var/downloadme.txt`. Our implementations would hash `/var/downloadme.txt` to yield an RI. Note that there are several ways a browser extension could calculate an RI. See Section 5 for a discussion of an alternative calculation based on Uniform Resource Name.

Next, we determine the so-called *Origin Domain* (OD). The OD is the base domain used to query the backend and should always be the Second-Level Domain (SLD) fragment of the *active browser tab's URL*, *i.e.,* the URL of the tab that initiated the download. For example: `somesite.com` would be the OD for a Chrome tab at URL `frag.something.somesite.com` and `fakesite.io` would be the OD for a Chrome tab at `git.fakesite.io`.

The OD is appended to the Primary Label (PL), which is then appended to the RI Sub-Label (SL). The PL is a standard string used to more easily identify the backend records our implementations rely upon; we used "_dnschk". It will always appear as the third-level domain following the OD in any query to the backend. The SL is a standard string used to identify backend entries that contain RIs; we used "_ri" in our implementations. The resulting construction, consisting of `SL.PL.OD` (*e.g.,* `_ri._dnschk.fakesite.io`), is appended to the RI calculated earlier. This forms the subject of the query to our backend, whereafter the backend responds with the AC or an indication that the RI-to-AC mapping was not found.

To remain in compliance with DNS protocol label limits, we chose to split the RI—a 64 character alphanumeric string—into two labels separated by a period. We do this for both implementations, though it is only relevant with DNSCHK.

The ultimate query sent to the backend consists of an OD (*e.g.,* `fakesite.io`), a PL (static; *i.e.,* `_dnschk`), an SL (static; *i.e.,* `_ri`), and an RI broken into two parts (*i.e.,* `RI1` and `RI2`)). This yields the following (with an example on line 2):

```
RI1.RI2.SL.PL.OD
RI1.RI2._ri._dnschk.fakesite.io
```

Finally, the backend responds to our query and NAC Validation is performed. If NAC Validation succeeds, our extensions render a "safe" judgement via the extension UI. If NAC Validation fails, our extensions render an "unsafe" judgement. If the backend response indicates the RI mapping we queried does not exist, there are two possible outcomes: the extensions render a "neutral" judgement if they are not operating under strict mode conditions, otherwise an "unsafe" judgement is rendered (as if NAC Validation had taken place and failed).

"Strict mode" status, if active, prevents DNSCHK and DHTCHK from rendering "neutral" judgements for a particular OD. The point of neutral judgements is to allow the HASCHK approach to be incrementally adopted and deployed on the open internet without "breaking the internet" or pestering the end user with false positives when downloading resources that are not explicitly protected by our approach. For resources that are protected by our approach, strict mode ensures there are only two possible judgements rendered by DNSCHK and DHTCHK for a given OD: either "safe" if NAC Validation succeeds or "unsafe" in all other circumstances. For this reason, it is recommended that any adopter of our approach ensure their resources are protected under strict mode by default.

To determine if strict mode status applies to an OD, an additional backend query is made of the form `SML.PL.OD`, where SML is the Strict Mode Sub-Label consisting of the standard string "_smode". Continuing with our previous example, our query would take the form `_smode._dnschk.fakesite.io`. If and only if the subject of this additional query exists in the backend, strict mode status is assumed.

3.4.2 Determining Origin Domain in the Browser

To execute a resource integrity attack on a web server, an attacker generally has two paths. They can mutate the resource in place, which would cause NAC Validation to fail. They could also mutate the web page hosting the resource, replacing the resource anchor with a malicious one that points to a compromised resource on the attacker's remote system. DNSCHK and DHTCHK will catch this due to how we calculate the Origin Domain (OD).

To prevent such implementation-specific attacks, we make a distinction between the domain a resource's hyperlink might be pointing to (which might belong to the attacker) and the OD, which is the domain of the web document that contains said hyperlink. The scope of the OD is at the tab level, meaning there is one OD determined for each open browser tab.

In our proof-of-concept implementations, we rely on the Chrome Tabs and WebRequest APIs to associate downloads with ODs. Our extensions are implemented such that the OD is determined as early as possible in a Chrome tab's navigation lifecycle. Further, in our implementations, determined ODs are ultimately ordered as a LIFO construction. This ensures the resource-to-tab mappings remain accurate in the case where two tabs share the same OD when a resource download is observed.

4 Evaluation

The primary goal of any HASCHK implementation is to alert end-users when the resource they have downloaded is something other than what they were expecting. In this section, we evaluate our approach by first assessing the threat model HASCHK addresses, followed by an examination of our proof-of-concept Google Chrome extensions, DNSCHK and DHTCHK. We then test our implementations versus a real-world deployment of HotCRP and/or a random sampling of papers published in previous USENIX Security proceedings. Finally, we evaluate the obstacles to scalability and potential performance overhead of our extensions.

4.1 The Threat Model

4.1.1 Compromised Resource

We consider the case where an attacker can influence or even completely control the victim's resource distribution mechanism (web page, file server, CDN, etc) in any way. In this context, the attacker can trick the user into downloading a compromised resource of the attacker's choice. This attack can be accomplished by compromising the resource on a victim system or tricking the user into downloading a compromised resource on the attacker's remote system.

In either case, the attacker does not have control over the backend system responsible for mapping RIs to ACs relevant to the function of HASCHK.

If the attacker does not alter the RI, the compromised resource will fail integrity validation during the NAC Validation step.

If the attacker does alter the RI, there are two possibilities: 1) the new RI *does not* exist in the backend, in which case HASCHK will fail to resolve the NAC, hence the NAC Validation step will fail; 2) the "compromised" RI *does* exist in the backend, therefore the RI must be pointing to a different resource's checksum.

In the first case, there are two further possibilities: a) NAC validation fails and HASCHK *is not* in strict mode, so a "neutral" judgement is rendered; b) NAC validation fails and HASCHK *is* in strict mode, so an "unsafe" judgement is rendered, warning end users that the resource is likely compromised.

In the second case, unless the attacker's goal is to swap one or more resources protected by HASCHK and a particular backend with another resource also protected by HASCHK and the same backend, the NAC Validation step will fail. For such a "swap" to work, the attacker would be required to both change the RI and also offer to the victim the HASCHK-protected resource the "compromised" RI corresponds to, which shrinks the attack surface here significantly.

4.1.2 Compromised Authoritative Checksum

We consider the case where an attacker can completely control the highly available backend that allows HASCHK to function. In this context, the attacker can return an authoritative response of their choice to any query.

In this case, the attacker does not have control over the victim's resource distribution mechanism (web page, file server, CDN, etc).

Even if the attacker achieved this egregious level of compromise, they do not have the ability to deliver a malicious payload in this case. However, the attacker could use control over the relevant backends to cause denial-of-service style attacks against those attempting to download the resource by causing all NAC Validation checks to fail. This is mitigated by HASCHK allowing the user to "override" its error states; *i.e.,* HASCHK does not mutate or quarantine downloaded resources. See Section 5 for further discussion on limitations due to the Chrome/WebExtensions API.

4.1.3 Compromised Resource and Authoritative Checksum

We consider the case where an attacker can influence or even completely control the victim's resource distribution mechanism (web page, file server, CDN, etc) in any way. Additionally, the attacker can completely control the victim highly available backend that allows HASCHK to function. Therefore, the attacker can make the user download a compromised resource and also return a (compromised) AC that legally corresponds to said compromised resource.

4.2 Real-World Resource Corruption Detection

To further evaluate the effectiveness of our mitigation, we test DNSCHK and DHTCHK—our proof-of-concept HASCHK Chrome extension implementations—against a series of common real-world resource integrity violations. The impetus behind any such resource integrity SCA is to have the resource pass through undetected by abusing the trust between client and provider with the hope that an unsuspecting user will interact with it.

We show that the HASCHK approach is more effective than existing approaches at detecting integrity violations in arbitrary resources on the internet; this is especially evident when DNSCHK and DHTCHK are compared to the de facto standard: checksums.

4.2.1 DNSCHK

To empirically evaluate DNSCHK, we launch a heavily modified version of the popular open source research submission and peer review software, *HotCRP* (version 2.102). Our modifications allow anyone visiting the site to interactively corrupt submissions and manipulate relevant DNS entries at will.

For our evaluation, we upload 10 different USENIX Security PDFs to our HotCRP instance. Upon their upload, HotCRP calculated and displayed the unique checksum (a SHA-256 digest) of each PDF. After each PDF is uploaded, we immediately download it and manually calculate a local checksum, matching each to the checksum displayed by the HotCRP software. Next, we utilize the custom functionality we added to our HotCRP instance to populate our DNS backend with each file's current "original" checksum. Each checksum is considered an Authoritative Checksum (AC) mapped to a Resource Identifier (RI) corresponding to the uploaded PDF item.

After installing DNSCHK into our Google Chrome browser, we again download each file. For each observed download, DNSCHK reported a "safe" judgement as expected. We then utilize the custom functionality we added to our HotCRP instance to add junk data onto the end of each of the uploaded PDFs, corrupting them. We also modified HotCRP so that it updated the displayed checksums to match their now-corrupted counterparts.

Once again, we download each file and calculate a local checksum. DNSCHK reported an "unsafe" judgement (a true positive) for each corrupted PDF file, as expected. Calculating the local checksum and checking it against the value reported by our HotCRP instance leads to a match (a false negative; *i.e.,* the result of co-location) as expected.

We ran this experiment three times and observed consistent results.

Finally, we implement a "redirection" attack where, when clicking the link to download the PDF document from HotCRP, users were forced to navigate to a "compromised" PHP script on an adjacent server that very quickly redirected clients several times before triggering the download of a corrupted version of the original HotCRP-hosted resource. Our implementation correctly flagged this download as suspicious once the download began, successfully warning the user.

4.2.2 DHTCHK

To evaluate DHTCHK, we connect to the authenticated global Ring OpenDHT network. Since the OpenDHT software is implemented in C++, it could not be included directly in a JavaScript plugin. Therefore, for our proof-of-concept implementation, we set up a local HTTP REST server wrapping the C++ implementation of OpenDHT. Our OpenDHT REST server provides an interface consistent with the one expected by DNSCHK (*i.e.,* Google DNS's REST API), allowing for code reuse (*e.g.,* redirection protection) between our two implementations. See Section 7, where we make these implementations available open source for community consideration.

For our evaluation, we manually calculate a local checksum (*i.e.,* an AC) and an RI for 10 different USENIX Security PDFs. For the OD here we use a static locally resolved domain name corresponding to a location on a local server. We manually store these RI-to-AC mappings as key-value pairs on the Kademlia-based OpenDHT network.

After installing DHTCHK into our Google Chrome browser, we download each file from our local server. For each observed download, DHTCHK reported a "safe" judgement as expected. We then store randomly generated checksums as replacement RI-to-AC mappings in the Ring OpenDHT network corresponding to these 10 PDFs such that the ACs purposely would not match the NACs generated by DHTCHK, simulating file corruption by an attacker.

Once again, we download each file and calculate a local checksum. DHTCHK reported an "unsafe" judgement for each corrupted PDF file, as expected.

4.3 Obstacles to Scalability, Deployment

As HASCHK is predicated on a distributed authenticated highly-available backend and requires no application/frontend source code changes to function, we conclude that the scalability of HASCHK can be reduced to the scalability of its backend. We are aware of no other obstacles to scalability beyond those inherited from the underlying backend system.

In respect to DNS specifically, packet fragmentation can be a concern for high performance networks [16], but this is an artifact of DNSSEC and related protocols rather than HASCHK [1]. Further, we are aware of no practical limits or protocol-based restrictions on the scalability of a backend file itself or its sub-zones. A service can host tens of thousands of resource records in their backend file [36, 37].

With the HotCRP demo, the totality of our resource deployment scheme consisted of the addition of a new TXT entry to our backend file—accomplished via API call to Google DNS—during HotCRP's paper submission process. This new TXT entry consisted of a mapping between a RI and its corresponding AC.

We find a DNS record addition or update during the resource deployment process to be simple enough for service administrators to implement and presents no significant burden to deployment outside of DNS API integration into a development team or other entity's software deployment toolchain. For reference, we implemented the functionality that automatically adds (and updates) the DNS records mapping the ACs and RIs of papers uploaded to our HotCRP instance in under 10 lines of JavaScript.

We note that, in the case where an entity's content distribution mechanism relies on, for instance, a mirroring service, third-party CDN, et cetera *that randomly or disparately mangles resource URL paths*, our proof-of-concept implementations currently require each "mangled" RI permutation representing a resource to be added to the backend, even if they all represent the same resource by a different name/path. This issue and its solutions are discussed further in the context of URNs in Section 5.

4.4 Performance Overhead

While evaluating our HASCHK implementations, we observe no additional network load or CPU usage with the extension loaded into Chrome. Measurements were taken using the Chrome developer tools. Intuitively, this makes sense since our HASCHK implementations make at most two queries to the backend before rendering a judgement. Hence, we find that HASCHK introduces no additional performance overhead. Further, as our implementations of HASCHK do not interrupt or manipulate resources as they are being downloaded, there is no additional download latency introduced by HASCHK.

4.5 Attacking OD Resolution in the Browser

While determining ODs, a clever attacker might attempt to fool this process by redirecting users one or more times before triggering a direct download of a compromised resource. The redirects would allow an attacker to completely manipulate the OD, with the ability to trick an unsuspecting user into downloading a compromised resource with valid entries in the backend system.

We mitigate this threat by leveraging JavaScript's document-wide *trusted event* [19] delegation capability. Specifically, when a tab navigation event is observed, the tab is flagged `suspicious` by default and the determined OD is not updated (*i.e.,* it remains at its previous value). If the user interacts with the tab (*i.e.,* a trusted click or key press event is observed after navigation completes), the `suspicious` flag is cleared and the OD is updated. If another tab navigation event occurs without user interaction first (*e.g.,* a quick redirect), this process repeats recursively. If a download is observed coming from a tab flagged `suspicious`, the user is warned about the suspicious circumstances similarly to receiving an "unsafe" judgement.

While this mitigates the attack as described, it has the side effect of potentially generating false positive warnings when 1) the user is redirected to a legitimate website—such as a download mirroring service—that automatically triggers a download after some amount of time when also 2) the user does not interact with the page at all before the download begins. We argue such cases are non-average and the tradeoff here is worthy.

5 Discussion

In this section, we examine current and previous DNS-based and other approaches to problems related to HASCHK. We specifically note PGP's limiting human factors, how those factors similarly apply to the application of checksums for resource integrity validation, and how our DNSCHK implementation avoids these factors. Thereafter, we discuss some limitations of our implementations.

5.1 Additional Related Work

Cryptographic Data in DNS Resource Records.

The DNS-Based Authentication of Named Entities (DANE) specification [20, 31, 54] defines the "TLSA" and "OPENPGPKEY" DNS resource records to store cryptographic data. These resource record types, along with "CERT" [33], "IPSECKEY" [48], those defined by DNS Security Extensions (DNSSEC) [1], and others demonstrate that storing useful cryptographic data retrievable through the DNS network is feasible at scale. Due the unique requirements of DNSCHK, however, we use "TXT" records to map Resource Identifiers to Authoritative Checksums. In accordance with RFC 5507 [21], a production DNSCHK implementation would necessitate the creation of a new DNS resource record type as no current resource record type meets the requirements of DNSCHK.

PGP/OpenPGP. Though PGP addresses a fundamentally different authentication-focused threat model compared with HASCHK, it is useful to note: many of the same human and UX factors that make the cryptographically solid OpenPGP standard and its various implementations so unpleasant for end users also exist in the context of download integrity verification and checksums. End users cannot and *will not* be burdened with manually verifying a checksum; as was the case with PGP 5.0 [52], some users are likely confused by the very notion and function of a checksum, if they are aware that checksums exist at all. If PGP's adoption issues are any indication, users of a security solution that significantly complicates an otherwise simple task are more likely to bypass said solution rather than be burdened with it. To assume otherwise can have disastrous consequences [52] (cf. Section 2).

Link Fingerprints and Subresource Integrity. The Link Fingerprints (LF) draft describes an early HTML anchor and URL based resource integrity verification scheme [35]. Subresource Integrity (SRI) describes a similar production-ready HTML-based scheme designed with CDNs and web assets (rather than generic resources) in mind. Like HASCHK, both LF and SRI employ cryptographic digests to ensure no changes of any kind have been made to a resource [2]. Unlike HASCHK, LF and SRI rely on the server that hosts the HTML source to be secure; specifically, the checksums contained in the HTML source must be accurate for these schemes to work. An attacker that has control of the web server can alter the HTML and inject a malicious checksum. With HASCHK, however, an attacker would additionally have to compromise whichever authenticated distributed system hosted the mappings between Resource Identifiers and Authoritative Checksums.

Content-MD5 Header. The Content-MD5 header field is a deprecated email and HTTP header that delivers a checksum similar to those used by Subresource Integrity. It was removed from the HTTP/1.1 specification because of the inconsistent implementation of partial response handling between vendors [22]. Further, the header could be easily stripped off or modified by proxies and other intermediaries [38].

Reproducible Builds/Deterministic Build Systems. A deterministic build system is one that, when given the same source, will deterministically output the same binary on every run. For example, many packages in Debian [46] and Arch Linux can be rebuilt from source to yield an identical byte for byte result each time via a reproducible build process [45]. When a deterministic build system is coupled with the HASCHK approach, a chain of trust can be established that links the *Development* and *Integration* supply chain phases to the *Deployment*, *Maintenance*, and *Retirement* supply chain phases (cf. Table 1), further raising the bar for the attacker.

5.2 Implementation-specific Limitations

5.2.1 DNSSEC Adoption is Slow

The proof-of-concept DNSCHK implementation is only secure if the corresponding DNS zone is secure, *i.e.,* it is protected by DNSSEC. As detailed in Fig. 1, DNSSEC adoption rate—which has increased dramatically since the first production root zone was signed in 2010 [6, 32]—is decidedly variable and slow to rise. Only around 3% of Fortune 1000 and 9% of university domains have properly deployed DNSSEC [41], and the number of DNS resolvers validating DNSSEC replies currently sits at approximately 12-14% [6].

While this could be happening for a variety of reasons [5,

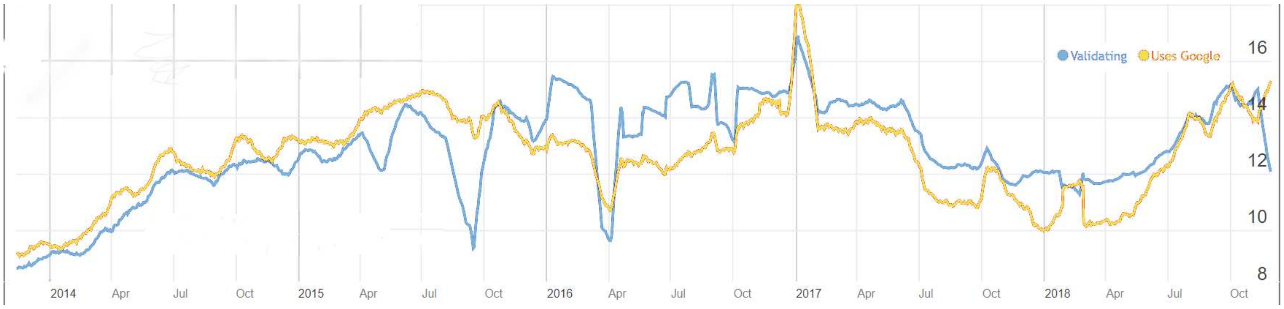

Figure 1: APNIC estimate of the percentage of global DNS resolvers (Google PDNS as well as local resolvers) performing DNSSEC validation from October 2013 to December 2018. The five year trend is positive.

15, 29, 30, 55], slow growth is certainly not outside of the norm for global protocol deployments that are perceived as "nice to have" rather than "business critical". For instance: the adoption rate of IPv6, proposed nearly 25 years ago, is similarly slow to rise. Globally, when measured as the availability of IPv6 connectivity among users accessing any Google service, it rests at approximately 21% [25], with only 2% of Fortune 1000 and 3% of university domains being IPv6-enabled [41]; we note that this is the case despite IANA and all RIRs having entered the final stage of virtual IPv4 address space exhaustion as of 2018 [53] while the number of internet-connected devices continues its upward climb [10].

With that said, if we assume the user has installed the DNSCHK extension, slow adoption of DNSSEC globally would in no way impact an individual entity's ability to adopt and immediately benefit from DNSCHK. We consider this a key feature of the approach. Those entities that consider their resources' integrity to be business critical do not have to wait for DNSSEC to be adopted globally. Any well-configured DNSSEC-protected zone can opt-in to providing the resource records DNSCHK expects, including a strict mode record, offering power users verifiable resources while remaining completely transparent to everyone else. Otherwise, users receiving resources from an entity that is not yet DNSSEC capable (and so they do not support DNSCHK) will not experience any interruption in their user experience whether they have a DNSCHK-capable browser or not.

5.2.2 DNS-Specific Protocol Limitations

Clearly, DNSCHK relies on DNS. However, DNS [36] was not originally designed to transport or store relatively large amounts of data, though this has been addressed with EDNS0 [16]. The checksums stored in DNS should not be much longer than 128 bytes or the output of the SHA512 function. Regardless, DNS resource record extensions exist that store much more than 128 bytes of data [31, 33, 48, 54].

Several working groups are considering DNS as a storage medium for checksums/hash output as well, such as securitytxt [23]. A widely deployed example of DNS "TXT" resource records being used this way is SPF and DKIM [12]. We are unaware of any practical limitation on the number of resource records a DNS zone file can support [36], hence any considerations regarding zone file size and/or ceilings on the number of TXT records in a single zone are at the sole discretion of the implementing entity.

Additionally, DNSCHK does not add to the danger of amplification and other reflection attacks on DNS; these are generic DNS issues addressable at other layers of the protocol.

5.2.3 DHT-Specific Limitations

Unlike DNS, an entity seeking to leverage a Distributed Hash Table (DHT) may not have the benefit of being able to rely on a high availability distributed authenticated backend that is already established, is well-tested, and exists globally like DNS. Such an entity would have to either maintain their own network of DHT nodes, which can incur significant cost if such a network was not already deployed, or piggy back off an open authenticated network, as was done with the case with our proof-of-concept DHTCHK implementation.

These reasons make DNS with DNSSEC more appealing as an authenticated backend in comparison. Regardless, we provide DHTCHK to demonstrate the utility and flexibility of the HASCHK approach.

5.2.4 Limitations of a Chrome Extension

Our current JavaScript proof-of-concept implementations, as Chrome extensions, are not allowed to touch the resource files downloaded by Chrome and so cannot prevent a potentially-malicious resource from being executed by the end user—a feature Chrome/Chromium reserves for its own internal use. The Chrome *app* API [24] might have been of assistance as it allowed for some limited filesystem traversal via a now deprecated native app API; there is also a non-standard HTML5/WebExtensions FileSystem API that would provide similar functionality were it to be widely considered [7].

While still effective, DNSCHK and DHTCHK would be even more effective as browser extensions if Chrome/Chromium or the WebExtensions API allowed for an explicit `onComplete` event hook in the downloads API.

This hook would fire immediately before a file download completed and the file became executable, *i.e.,* had its `.crdownload` or `.download` extension removed. The hook would consume a `Promise`/`AsyncFunction` that kept the download in its non-complete state until said `Promise` completed. This would allow the extensions' background pages to do something like alter a download's `DangerType` property and alert the end user to a dangerous download naturally. These modifications would have the advantage of communicating intent through the browser's familiar UI and preventing the potentially-malicious download from becoming immediately executable. Unfortunately, the closest the Chrome/WebExtensions API comes to allowing `DangerType` mutations is the `acceptDanger` method on the downloads API, but it is not suitable for use with DNSCHK as a background page based extension.

While nice to have, we stress that none of the aforesaid functionality is critical to the ability of our implementations to more effectively mitigate SCA risk than checksums and other solutions (cf. Section 4).

5.3 URNs vs RIs for Wider Compatibility

The goal of Resource Identifiers (RI) and Non-Authoritative Checksums (NAC) is very similar to that of Uniform Resource Names (URN). It may make sense to replace the mapping between NACs/RIs and Authoritative Checksums with purely URN-based DNS lookups that return specially formatted TXT records upon success. This would further simplify the deployment process for service administrators since DNS updates would be based upon the resource's contents instead of both its contents *and where it is located physically on a distribution system*. It may also allow for additional confirmation methods of the identical resources in different domains and in different locations.

We did not create a URN-based scheme in our initial approach due to a new URN scheme requiring the registration of a unique identifier with the Internet Assigned Numbers Authority. Going forward, we can potentially adopt a URN scheme that already exists, such as Magnet links [34] or the informal IETF draft for hash-based URN namespaces [50]. URNs would allow DNSCHK and DHTCHK to be much more flexible in how they map resources to Authoritative Checksums, *i.e.,* our proof-of-concept implementations would be able to handle mirrors and CDNs that may mangle resource path.

6 Conclusion

Downloading resources over the internet is indeed a risky endeavor. Resource integrity attacks, and Supply Chain Attacks more broadly, are becoming more frequent and their impact more widely felt. This paper shows that the de facto standard for addressing resource integrity risk—the use of *checksums* coupled with a secure transport layer—is an insufficient and often ineffective solution. We propose a novel practical resource validation approach meant as a complete replacement for checksum based approaches: HASCHK, which automates the tedious parts of verification to eliminate user apathy while leveraging highly-available authenticated distributed systems to ensure resources and checksums are not co-hosted. Further, we demonstrate the effectiveness and practicality of our approach versus resource integrity attacks in a real-world system.

The results of our evaluation show that our approach is more effective than checksums and other attempts at mitigating resource integrity attacks for arbitrary resources on the internet. Further, we show DNSCHK and DHTCHK are capable of detecting a wide variety of real-world integrity errors, significantly raising the bar for the attacker. DNSCHK, as it is backed by DNS, is deployable at scale for entities that already choose to secure their DNS zone(s) with DNSSEC; this can be done without fear of adversely affecting the user experience of non-compliant clients.

7 Availability

We make our DNS[1] and DHT[2] proof-of-concept implementations of HASCHK available to the community open source so that others can extend it or compare to it. Our hope is that this work motivates further exploration of resource integrity and other SCA attack mitigation strategies.

We also make publicly available for your consideration our testing environment: a patched HotCRP instance[3].

[1] DNSCHK: `https://tinyurl.com/dnschk-actual`

[2] DHTCHK: `https://tinyurl.com/dhtchk-actual`

[3] HotCRP Testbed: `https://tinyurl.com/dnschk-hotcrp`

References

[1] D. E. E. 3rd. *Domain Name System Security Extensions*. RFC 2535. http://www.rfc-editor.org/rfc/rfc2535.txt. RFC Editor, 1999. URL: http://www.rfc-editor.org/rfc/rfc2535.txt.

[2] D. Akhawe, F. Braun, J. Weinberger, and F. Marier. *Subresource Integrity*. W3C Recommendation. http://www.w3.org/TR/2016/REC-SRI-20160623/. W3C, 2016.

[3] D. Akhawe and A. P. Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness." In: *USENIX security symposium*. Vol. 13. 2013.

[4] T. Anderson. "Linux Mint hacked: Malware-infected ISOs linked from official site". In: *The Register* (2016). URL: https://www.theregister.co.uk/2016/02/21/linux_mint_hacked_malwareinfected_isos_linked_from_official_site.

[5] APNIC. *Why DNSSEC deployment remains so low*. 2017. URL: https://blog.apnic.net/2017/12/06/dnssec-deployment-remains-low/.

[6] APNIC. 2018. URL: https://stats.labs.apnic.net/dnssec/XA?c=XA&x=1&g=1&r=1&w=7&g=0.

[7] A. Barstow and E. U. *Re: [fileapi-directories-and-system/filewriter]*. 2014. URL: http://lists.w3.org/Archives/Public/public-webapps/2014AprJun/0012.html.

[8] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. *Transport Layer Security (TLS) Extensions*. RFC 3546. RFC Editor, 2003.

[9] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *OpenPGP Message Format*. RFC 4880. http://www.rfc-editor.org/rfc/rfc4880.txt. RFC Editor, 2007. URL: http://www.rfc-editor.org/rfc/rfc4880.txt.

[10] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper*. Tech. rep. Cisco, 2017. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[11] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. http://www.rfc-editor.org/rfc/rfc5280.txt. RFC Editor, 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[12] D. Crocker, P. Hallam-Baker, and T. Hansen. *DomainKeys Identified Mail (DKIM) Service Overview*. RFC 5585. 2009. DOI: 10.17487/RFC5585. URL: https://rfc-editor.org/rfc/rfc5585.txt.

[13] *CVE-2012-5159*. Available from MITRE, CVE-ID CVE-2012-5159. 2012. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5159.

[14] J. Daemen and V. Rijmen. "AES proposal: Rijndael". In: (1999).

[15] T. Dai, H. Shulman, and M. Waidner. "Dnssec misconfigurations in popular domains". In: *International Conference on Cryptology and Network Security*. Springer. 2016, pp. 651–660.

[16] J. Damas, M. Graff, and P. Vixie. *Extension Mechanisms for DNS (EDNS(0))*. STD 75. RFC Editor, 2013.

[17] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246. http://www.rfc-editor.org/rfc/rfc2246.txt. RFC Editor, 1999. URL: http://www.rfc-editor.org/rfc/rfc2246.txt.

[18] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. http://www.rfc-editor.org/rfc/rfc5246.txt. RFC Editor, 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[19] *DOM*. URL: https://dom.spec.whatwg.org/#dom-event-istrusted.

[20] V. Dukhovni and W. Hardaker. *The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance*. RFC 7671. RFC Editor, 2015.

[21] P. Faltstrom, R. Austein, and P. Koch. *Design Choices When Expanding the DNS*. RFC 5507. RFC Editor, 2009.

[22] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://www.rfc-editor.org/rfc/rfc7231.txt. RFC Editor, 2014. URL: http://www.rfc-editor.org/rfc/rfc7231.txt.

[23] E. Foudil and Y. Shafranovich. *A Method for Web Security Policies*. Internet-Draft draft-foudil-securitytxt-04. http://www.ietf.org/internet-drafts/draft-foudil-securitytxt-04.txt. IETF Secretariat, 2018. URL: http://www.ietf.org/internet-drafts/draft-foudil-securitytxt-04.txt.

[24] Google. *Chrome APIs*. URL: https://developer.chrome.com/apps/api_index.

[25] Google. *Google IPv6 Statistics*. URL: https://www.google.com/intl/en/ipv6/statistics.html.

[26] HandBrake. *Mirror Download Server Compromised*. 2017. URL: https://forum.handbrake.fr/viewtopic.php?f=33&t=36364.

[27] "HandBrake hacked to drop new variant of Proton malware". In: *MalwareBytes Labs* (2017). URL: https://blog.malwarebytes.com/threat-analysis/mac-threat-analysis/2017/05/handbrake-hacked-to-drop-new-variant-of-proton-malware/.

[28] "Havex". In: *New Jersey Cybersecurity and Communications Integration Cell Threat Profiles* (2017). URL: https://www.cyber.nj.gov/threat-profiles/ics-malware-variants/havex.

[29] A. Herzberg and H. Shulman. "DNSSEC: Security and availability challenges". In: *Communications and Network Security (CNS), 2013 IEEE Conference on*. IEEE. 2013, pp. 365–366.

[30] A. Herzberg and H. Shulman. "Towards Adoption of DNSSEC: Availability and Security Challenges." In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 254.

[31] P. Hoffman and J. Schlyter. *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698. http://www.rfc-editor.org/rfc/rfc6698.txt. RFC Editor, 2012. URL: http://www.rfc-editor.org/rfc/rfc6698.txt.

[32] S. Isasi. *Expanding DNSSEC Adoption*. 2018. URL: https://blog.cloudflare.com/automatically-provision-and-maintain-dnssec/.

[33] S. Josefsson. *Storing Certificates in the Domain Name System (DNS)*. RFC 4398. RFC Editor, 2006.

[34] *MAGNET-URI Project*. URL: http://magnet-uri.sourceforge.net/.

[35] G. Markham. *Link Fingerprints*. 2008. URL: http://www.gerv.net/security/link-fingerprints/.

[36] P. Mockapetris. *Domain names - concepts and facilities*. STD 13. http://www.rfc-editor.org/rfc/rfc1034.txt. RFC Editor, 1987. URL: http://www.rfc-editor.org/rfc/rfc1034.txt.

[37] P. Mockapetris. *Domain names - implementation and specification*. STD 13. http://www.rfc-editor.org/rfc/rfc1035.txt. RFC Editor, 1987. URL: http://www.rfc-editor.org/rfc/rfc1035.txt.

[38] J. Myers and M. Rose. *The Content-MD5 Header Field*. RFC 1864. RFC Editor, 1995.

[39] N. Nelson. "The impact of dragonfly malware on industrial control systems". In: *SANS Institute* (2016).

[40] NIST. *Cyber Supply Chain Risk Management*. URL: https://csrc.nist.gov/Projects/Supply-Chain-Risk-Management.

[41] NIST. *Estimating IPv6 and DNSSEC Deployment Status*. URL: https://usgv6-deploymon.antd.nist.gov/govmon.html.

[42] NIST. *Software Supply Chain Attacks*. 2017. URL: https://csrc.nist.gov/csrc/media/projects/supply-chain-risk-management/documents/ssca/2017-winter/ncsc_placemat.pdf.

[43] "phpMyAdmin corrupted copy on Korean mirror server". In: *SourceForge Official Blog* (2012). URL: https://sourceforge.net/blog/phpmyadmin-back-door/.

[44] "PMASA-2012-5". In: *PMASA-2012-5* (2012). URL: https://www.phpmyadmin.net/security/PMASA-2012-5/.

[45] *Reproducible Builds: A Set Of Software Development Practices That Create An Independently Verifiable Path From Source To Binary Code*. URL: https://reproducible-builds.org/.

[46] *Reproducible Builds (Debian Wiki)*. URL: https://wiki.debian.org/ReproducibleBuilds.

[47] E. Rescorla. *HTTP Over TLS*. RFC 2818. http://www.rfc-editor.org/rfc/rfc2818.txt. RFC Editor, 2000. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

[48] M. Richardson. *A Method for Storing IPsec Keying Material in DNS*. RFC 4025. RFC Editor, 2005.

[49] L. M. Team. *Beware of hacked ISOs if you downloaded Linux Mint on February 20th!* 2016. URL: https://blog.linuxmint.com/?p=2994.

[50] P. Thiemann. *A URN Namespace For Identifiers Based on Cryptographic Hashes*. Internet-Draft draft-thiemann-hash-urn-01. Archived. Internet Engineering Task Force, 2003. 10 pp. URL: https://datatracker.ietf.org/doc/html/draft-thiemann-hash-urn-01.

[51] Z. Whittaker. "Hacker explains how he put "backdoor" in hundreds of Linux Mint downloads". In: *ZDNet* (2016). URL: https://www.zdnet.com/article/hacker-hundreds-were-tricked-into-installing-linux-mint-backdoor/.

[52] A. Whitten and J. D. Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0". In: *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=1251421.1251435.

[53] P. Wilson. *How bad is IPv4 address exhaustion?* 2018. URL: https://blog.apnic.net/2018/02/15/bad-ipv4-address-exhaustion/.

[54] P. Wouters. *DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP*. RFC 7929. RFC Editor, 2016.

[55] Y. Yao, L. He, and G. Xiong. “Security and cost analyses of DNSSEC protocol”. In: *International Conference on Trustworthy Computing and Services*. Springer. 2012, pp. 429–435.