When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability

Scott Yilek
UC San Diego
syilek@cs.ucsd.edu

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

Hovav Shacham
UC San Diego
hovav@cs.ucsd.edu

Brandon Enright
UC San Diego
bmenrigh@ucsd.edu

Stefan Savage
UC San Diego
savage@cs.ucsd.edu

ABSTRACT

We report on the aftermath of the discovery of a severe vulnerability in the Debian Linux version of OpenSSL. Systems affected by the bug generated predictable random numbers, most importantly public/private keypairs. To study user response to this vulnerability, we collected a novel dataset of daily remote scans of over 50,000 SSL/TLS-enabled Web servers, of which 751 displayed vulnerable certificates. We report three primary results. First, as expected from previous work, we find an extremely slow rate of fixing, with 30% of the hosts vulnerable when we began our survey on day 4 after disclosure still vulnerable almost six months later. However, unlike conventional vulnerabilities, which typically show a short, fast fixing phase, we observe a much flatter curve with fixing extending six months after the announcement. Second, we identify some predictive factors for the rate of upgrading. Third, we find that certificate authorities continued to issue certificates to servers with weak keys long after the vulnerability was disclosed.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General —*Security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.3 [**Computer-Communication Networks**]: Network Operations

General Terms

Measurement, Security

Keywords

Debian, OpenSSL, PRNG, entropy, attacks, survey

1. INTRODUCTION

OpenSSL is a commonly-used cryptographic library with related command-line tools. Beginning in September, 2006, the package for OpenSSL included in the Debian distribution of Linux was modified to incorporate a bugfix intended to eliminate uninitialized memory reads flagged by the memory checking tool Valgrind. The bugfix did not just this but more: it eviscerated OpenSSL's entropy gathering. Until the problem was noticed by Luciano Bello [18] in May of 2008, the entropy available to applications running on Debian (and Debian-derived distributions, such as Ubuntu) was severely constrained. This vulnerability had a major impact on SSL/TLS and SSH servers. Each server possesses a public/private keypair, but any keypairs generated on an affected machine are easily predictable to an attacker. Knowledge of the private key allows an attacker to impersonate that server even when SSL/TLS or SSH is used and in many cases to undetectably decrypt traffic to and from the server.

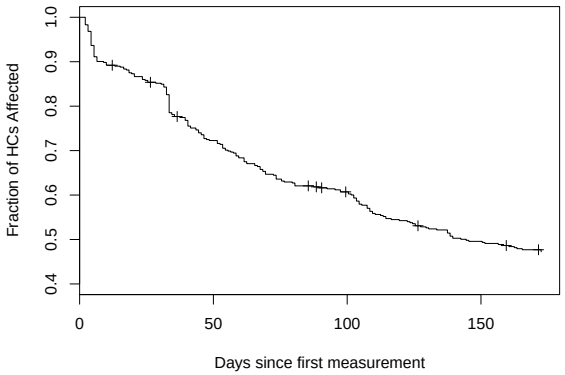

Figure 1: Overview of certificate updating

Recovery from this bug was more complicated than for a typical vulnerability. Patching affected machines, by itself, provided protection only against a small, less important class of attacks. Because the server's long-lived keypair was compromised, administrators needed to generate a new keypair and disseminate it to users. For SSL servers this typically required obtaining a new certificate for that keypair, a fairly heavyweight operation for certificates that aren't self-signed.

The goal of this work is to measure recovery from this type of vulnerability and compare it to what is known about recovery from other vulnerabilities. While it is infeasible to measure when servers are fixed, we can easily measure when they begin to display strong rather than weak public keys. We performed a daily survey of popular SSL servers,

IMC'09, November 4–6, 2009, Chicago, Illinois, USA.

beginning shortly after the bug was disclosed and continuing for some six months. Approximately 1.5% of those servers displayed weak keys and we were able to study the time course of fixing. As shown in Figure 1, the replacement of weak keys is a long, slow process, quite different from the fast fixing processes seen with typical vulnerabilities [15, 16]. [Hashmarks on the diagram indicate *censored* units, which stopped responding during the survey while still vulnerable.]

Our survey also yields new information about real-world SSL usage. Due to the bug's effects, we can determine, for each affected certificate, the architecture of the machine used to generate it and the process ID of the responsible process. This is the first time such data has been available. Even for the majority of sites unaffected by the bug, our data reveals how certificates of popular SSL sites are updated over time. Our collection of a new dataset by different methods allows us to reexamine previous work on SSL server demographics. We believe our dataset will be useful for other studies. See the Web page for this paper for information on obtaining a copy: `https://www-cse.ucsd.edu/groups/security/debiankey/`.

2. RELATED WORK

We build on two major previous lines of work: demographic surveys of SSL servers and longitudinal studies of vulnerability fixing.

SSL Server Surveys. There have been a number of previous surveys of the properties of SSL servers, mostly focusing on deployment of new versions of SSL, support for strong cryptographic algorithms, and valid third-party certificates. In 2000, Murray [12] surveyed 8081 SSL servers and found that around a third supported "weak" algorithms only (shockingly weak, in fact, by modern standards). He also found that around 10% of servers had expired certificates and around 3% had self-signed certificates. In 2005 and 2006 Lee [9] et al. repeated and expanded upon this work with a sample of 19,429 servers. They found that the situation had improved significantly; less than 5% of servers were weak by Murray's definition, although an uncomfortably high percentage of servers still supported the old "export" cipher suites (>90%) and SSL version 2 (>80%). They did not measure certificate validity.

Netcraft [13] runs a monthly survey attempting to cover all servers on the Internet. While this survey does not measure cipher suite support, Netcraft does collect information on certificate validity: they find that around 25% (estimated from their figures; raw numbers were not provided) of sites have self-signed certificates and less than half are from a "valid CA". Netcraft doesn't define this term but presumably it refers to one of the major CAs in the browser root list. Note that this data is very different from that reported by Murray, who found mostly valid certificates. We shed some light on this discrepancy in Section 7.1.

While our work is not specifically intended as a replication of either Lee et al.'s or Netcraft's research, we collect much of the same data as a side effect of our measurements. Section 7.1 describes some interesting contrasts.

Studies of Vulnerability Fixing. The topic of upgrading rate in response to vulnerabilities has been studied by Rescorla [16] and Ramos et al. [15]. The general pattern for externally visible "critical vulnerabilities" seems to be of a fast (half-life on the order of 10–20 days) exponential fixing phase immediately after the announcement of the vulnerability. The OpenSSL vulnerability studied by Rescorla provides probably the closest parallel: that study showed two rounds of patching, one in response to the initial vulnerability and one in response to the release of a worm. Each wave was relatively fast, with the vast majority of patching happening within two weeks and almost none after a month. Ramos reports a similar set of patterns as well as that some vulnerabilities show only irregular decline or no decline at all (this may be an artifact of survey methodology). The vulnerability we study exhibits yet a different pattern: a long, slow, flat, fixing cycle that actually accelerates in the first 30 days with significant levels of fixing as far out as six months. In Section 7.2 we provide some potential explanations for this difference.

3. BACKGROUND: SSL KEY EXCHANGE

In order to understand the issues discussed in this paper, it is necessary to have a basic understanding of how SSL works. In this section, we provide a brief overview of the relevant aspects of SSL. Although the purpose of SSL is to protect *data*, like many other cryptographic protocols such as SSH [20] and IPsec [4], it starts with a *handshake* phase which authenticates the peers and establishes joint keying material. That keying material is then used to protect the data flowing between the communicating peers. Figure 2 shows a highly simplified view of the most common variant of the full SSL handshake: "static RSA". [The technical term for the set of cryptographic algorithms used for a given SSL connection is *cipher suite*.]

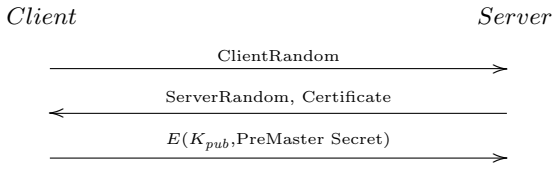

Figure 2: SSL static RSA handshake

In the static RSA version of SSL, the server generates a single, long-term RSA keypair (K_{pub}, K_{priv}) used for each transaction. In theory the server operator then acquires a certificate from a well-known *certificate authority* (CA) attesting to the binding between the RSA public key and the server's domain name (e.g, `www.amazon.com`). Any client that trusts the CA can then verify that binding. In some cases, however, the server acts as its own CA and generates a "self-signed" certificate. Such a certificate is just a key carrier: clients cannot verify the server's identity unless they have some independent channel for verifying the certificate.

When the client contacts the server, it sends a random nonce (the ClientRandom value). This is sent in the clear and is used solely to ensure uniqueness of the keying material for each connection. The server responds with its own ServerRandom value and a copy of its certificate. All of this information is also known to any observer. The client can then verify the server's certificate and extract K_{pub}. It then generates a random PreMasterSecret (PMS) value; encrypts that value under K_{pub} and sends the resulting EncryptedPreMasterSecret (EPMS) to the server. Because the server knows K_{priv} it can decrypt the EPMS to recover the PMS.

At this point, both the client and the server know the PMS, but any observer who doesn't know the server's private key does not. The PMS is then mixed with the client and server random values to form the keys which are used to encrypt traffic between client and server.

This exchange uses four random values: the server's RSA keypair, the client and server randoms, and the PMS. However, it's important to recognize that the client and server randoms need not be secret (and in fact only really need to be unique) and that the server's RSA keypair is not generated in real-time; only the PMS must be generated securely during the connection.

The other common full handshake variant, "ephemeral Diffie-Hellman" (DHE), is shown in Figure 3; it has a rather different set of properties. As before, the server has a long-term RSA key (sometimes this is a DSA key, but so rarely as to be irrelevant for the purposes of this discussion), but instead of having the client encrypt under that key, the client and the server do a Diffie-Hellman key exchange, with each side generating a new ephemeral DH key for the handshake. In order to authenticate the server side (as with static RSA, the client is not generally authenticated), the server signs its DH share with K_{priv}. Once the client has received and verified the server's share, it generates its own DH share and sends it to the server. The combined DH shared secret (often known as ZZ) is used as the PMS.

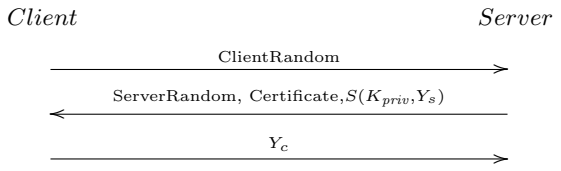

Figure 3: SSL ephemeral DH handshake

Unlike the static RSA mode, in DHE mode both sides need to generate strong random numbers. The security of Diffie-Hellman depends on the randomness of both sides' private keys; an attacker who can predict either side's DH private key can decrypt the connection. By contrast, an attacker who knows the server's RSA private key can impersonate the server but cannot passively decrypt connections which use DHE mode. This property is known as *Perfect Forward Secrecy* (PFS).

There is one final variant of SSL to consider, one in which neither DH nor RSA is used. Because DH and RSA operations are fairly computationally expensive, SSL incorporates a "session resumption" feature. The first time that a client and server pair communicate they establish a PMS which is then converted into a long-term MasterSecret (MS). The server provides the client with a "session id" which it can use to establish a new connection based on the same MS. The security of this resumed connection of course depends on the original handshake, but because the client and server random values are new, the connection will use different traffic keys to encrypt the actual data, thus protecting against a variety of cryptographic attacks (replay, cut-and-paste, etc.)

4. THE VULNERABILITY

The history of the Debian OpenSSL randomness vulnerability has been well-covered elsewhere [7, 6, 8], and we will not repeat it. Instead, we explain the technical details of the bug and its implications for SSL security.

4.1 Overview of the Bug

OpenSSL's *pseudorandom number generator* (PRNG), like all PRNGs, is a deterministic function: an attacker who knows all the inputs and the sequence of invocations can predict the output. To make the PRNG secure, the entropy pool must be *seeded* with many bytes from `/dev/random` or another source of entropy that an attacker cannot predict.

OpenSSL exposes two functions that update a program's entropy state: `RAND_add` and `RAND_bytes`.[1] The basic function used to update the entropy pool is `RAND_add`. `RAND_add` is called with a block of bytes b of length l. `RAND_add` then mixes all of these values into the PRNG entropy pool. The effect of the Debian bugfix is to modify `ssleay_rand_add` to mix in l but not b, with the effect that an attacker who knows the calling sequence (which is mostly determined by the program and not by the state of the machine) can predict the contents of the entropy pool and hence the output of the PRNG.

`RAND_bytes`, the function used to generate new random numbers, also updates the PRNG state with the number of bytes to be extracted and the program's process ID (pid) at the time the call is made. Folding in the current pid ensures that forked processes, which otherwise would have identical entropy pools, do not obtain the same values from the PRNG. (Reuse of random numbers renders many cryptographic protocols insecure.) It is the binary in-memory representation of the pid that is incorporated, so an attack must consider the endianness and native word size of the target machine. Because `RAND_bytes` folds in the number of bytes extracted, asking first for 20 bytes and then for 10 produces different output than if the calls are reversed.

Because a program's entropy pool starts in a known (all-zero) state, a remote attacker can track its evolution if he knows:

1. The sequence of calls to `RAND_add` and `RAND_bytes` made by the program.
2. For each call to `RAND_add`, the number of bytes to be added.
3. For each call to `RAND_bytes`, the number of bytes to be extracted and the program's process ID (pid) when the call is made.

This analysis holds true even if a program's behavior depends on the value of previous PRNG output, for example in the standard method for prime generation. When the PRNG is considered part of the program, the entire system is still deterministic given its initial state and its inputs.

4.2 The Effect of the Bug on SSL

The effect of this bug is contingent both on whether the client or server is affected and on which cipher suites are in use. If the client random number generator is broken, a passive attacker can usually predict the traffic keys (no matter what the cipher suite). In RSA mode, the attacker can predict the PMS (see, for example, Wagner and Goldberg [3]) and in DHE mode he can predict the client's DH private key, which allows prediction of the PMS (Abeni, Bello, and Bertacchini, demonstrate this attack on `DHE_RSA`

[1]These are wrappers around the `ssleay` functions below.

cipher suites for command-line clients and servers [1] affected by the Debian bug). However, as a practical matter the effect of this bug on clients is limited because most popular Web browsers do not use OpenSSL: Internet Explorer uses Microsoft's SChannel and Firefox uses NSS. Furthermore, Debian and other Linux distributions are not widely used as Web-browsing platforms. The most popular Unix browser based on OpenSSL is KDE's Konqueror, whose usage share — across all Unix platforms, not just Debian — is well under 0.05%.[2] However, many popular non-Web clients as well as command line Web clients such as `wget` use OpenSSL and therefore are likely to be affected.

Servers, on the other hand, represent a serious concern. OpenSSL is the dominant SSL implementation on server platforms, and Debian-derived distributions are popular on servers. There are two major avenues of attack: key generation and DH share generation.

RSA Key generation. If the server RSA keypair was generated on an affected version of OpenSSL, then the attacker can directly recover the private key.

The simplest and most common way to generate long-lived RSA keypairs for OpenSSL-based servers is to run the `openssl genrsa` program, invoked either directly or through a wrapper. This program uses the OpenSSL PRNG to generate the keys. As discussed in Section 4.1, each possible pid and platform configuration gives rise to a PRNG stream and thus to a unique RSA keypair. The attacker can pregenerate all those keypairs (this takes hours to days) and whenever one of the public keys matches he immediately knows the corresponding private key. Generating all possible keypairs is subtle; we give the details in Section 6.2.

Because the knowledge of the private key is all that differentiates the server from other entities, any attacker who knows the private key can impersonate the server. This attack can continue even after the server has replaced his key because the attacker still has a certificate/keypair and many clients do not check *certificate revocation lists* (CRLs). Moreover, if a static RSA cipher suite is used, the attacker can passively monitor connections and recover the PMS and therefore the traffic keys, thus gaining access to all the encrypted data as well as the ability to inject data of his choice. As discussed above, this latter attack is not possible with DHE cipher suites.

It's very important to realize that both of these attacks depend solely on the machine that the keypair was generated on; if that machine was affected by the bug but the SSL server itself is not (either because it is not a Debian machine or because it has been patched), attacks are still possible.

DHE Key Generation. By contrast, if the operational server is affected, then an attacker may be able to predict the server's DHE share even if the server's RSA keypair was securely generated. For a simple server that handles a single connection and then restarts — for example, an IMAP or POP server launched from `inetd` — then there would be a small number of possible values for the server's ephemeral private key X_s. The attacker can determine which of these possible keys is used for a connection by recognizing the ServerRandom and ephemeral public key values that accompany each. Unfortunately for the attacker, predicting these random values for real-world Web servers is more complicated than for the simple attack described by Abeni, Bello, and Bertacchini [1].[3]

Consider the case of the most popular Web server, Apache, which uses a "thundering herd" architecture with multiple long-lived worker processes. Each worker process handles multiple connections in sequence and, for each connection, calls `RAND_bytes` one or more times, mixing the entropy pool. Even with the Debian bug, the random values obtained by the process for the first connection it handles will be different from those obtained for the second and subsequent connections. What's more, the pattern of `RAND_bytes` invocations depends on the cipher suite and whether resumption is used. Thus, though we know the initial state of the server, we rapidly accumulate uncertainty about its sequence of `RAND_bytes` invocations. Unless an attacker can observe the entire set of connections from server startup, predicting the sequence of random values quickly becomes infeasible. Finally, each worker process has its own state and the attacker cannot directly measure what worker process is handling any given connection. Thus, even an attacker who can observe all server activity still must do a significant amount of work; we describe a full attack along these lines in Appendix A. This affects not only attackers who wish to recover a connection's PMS but also those who wish to fingerprint vulnerable servers using the predictable ServerRandom values they emit.

In contrast to the case where the server's key is weak, this avenue of attack is possible only when the client and server negotiate a `DHE_RSA` cipher suite. If the server's randomness is weak but a `RSA` cipher suite is chosen, the resulting connection is *entirely secure*, because it is the client that supplies the premaster secret.[4]

5. REMOTELY MEASURABLE DATA

As with other studies of this type, we collected data by remotely probing the server to determine its characteristics. Thus, the data we can report is limited to what we can collect via this mechanism.

As described above, servers can be affected by the bug we study in several ways: the server software can be affected, the server keypair can be weak, or both. Ideally we would like to be able to measure the evolution of both properties over time. This would allow us not only to replicate Rescorla's 2003 study [16] but also to measure a form of fixing that is related to but distinct from server software: Whereas previous papers have focused on the response to attack on servers, this paper illustrates the response to an attack on the data sent by servers. Unfortunately, our ability to measure remotely does not allow this. Determining the vulnerability of the server keypair is straightforward: construct a list of the weak server keys and check to see if the server's certificate contains such a key. In addition, when we determine that a server's key is vulnerable, this also gives us information about the machine on which the key was generated (which will often be the server); the exact set of keys is somewhat platform specific and so we can ex-

[2]See `http://marketshare.hitslink.com/report.aspx?qprid=1&qpcustom=Konqueror`.

[3]This was acknowledged by Bello in private communication, responding to an initial writeup of our analysis at `http://www.educatedguesswork.org/2008/08/the_debian_openssl_prng_bug_an.html`.

[4]Note that this is contrary to the naïve expectation that a protocol's security guarantees are destroyed when one party relies on predictable randomness.

tract the word size, endianness, and base OpenSSL version; see Section 6.2.

By contrast, remotely measuring the quality of the server software's PRNG (as opposed to the keypair) is not straightforward. We cannot directly examine the PRNG for the reasons described in Section 4.2 and although some Apache installations advertise the version of OpenSSL they are running, many Debian servers do not advertise this (it isn't entirely clear what the controlling factor is). Even if we could examine the version name, because the error was in the Debian fork of OpenSSL, the OpenSSL version number is not diagnostic here. Thus, we cannot reliably determine the status of the server itself.

As Murray [12] and Lee et al. [9] show, it is possible to determine what parameters a server is willing to negotiate by probing it using a client with a limited set of options. Because our interest is primarily limited to bug fix deployment, our survey was less exhaustive (and less intrusive) but we still measure a number of the same parameters, and in particular what cipher suite the server will select when offered the default set from OpenSSL.

This bug also affected SSH servers, and it is possible to remotely determine whether they have weak keys [11] — source code examination suggests that it may also be possible to determine the status of the server but we have not yet tried that. We initially developed SSH probing tools as well, but ultimately decided to measure only SSL servers. Our primary reason was logistical: SSL server operators expect connections to their servers from arbitrary sources. By contrast, unexpected connections to SSH servers are often perceived as an attack. Indeed, the only complaint we received in our survey was from the operator of an SSH server that had been inadvertently included on our probe list. In addition, because any given SSH server is used only by a relatively small number of users, it is less clear what a representative sample would look like.

6. METHODOLOGY

In the remainder of the paper, we describe our survey of SSL servers. Because only a small fraction of servers were likely to run an affected version of Linux, we first needed to collect a large set of servers to sample. Drawing up a list of representative SSL servers is not easy. A random scan of the IP space would be as likely to happen upon an unused "You have successfully installed Apache" site as PayPal's servers. (Because the fraction of IPs serving content on TCP port 443 is low, such a scan would also be intrusive.) Furthermore, while there exist lists, such as Alexa's, of popular Websites, the popularity of a site is a poor proxy for the popularity of its associated secure SSL site, if there is one. Many popular sites, such as the Drudge Report, serve a substantial amount of traffic over HTTP and none over HTTPS. We chose to use measured SSL usage as a selection procedure. Through the UC San Diego Information Security Office, which routinely monitors UCSD network usage, we were able to obtain a list of all IP addresses to which a 1 KB or larger flow of traffic on TCP port 443 had been detected in the 56-day period ending 21:00 UTC on Friday, May 16, 2008. This list contained 59100 servers.

Because our list of SSL server addresses consists of servers actually visited by a diverse user population of a large organization, we believe that it is more representative of SSL as deployed on the Internet than a random scan would be. We expect that the corpus of data we collected about these servers will be of wider use.

6.1 Data Collection

Using this list as our starting point, we constructed a simple program which would attempt to contact each server on the list (directly connecting the IP address with no DNS lookup) and initiate an SSL handshake using the "`openssl s_client -connect`" command. If the handshake did not complete within 30 seconds, we marked the host as failed and moved on.[5] The results for each run were then stored in raw form to a separate file for that host and day. The output is simply the output of OpenSSL, which includes: the negotiated SSL protocol version; the server certificate chain; the selected cipher suite; the session ID; the computed master key; and the start time of the connection.

Starting on the evening of Saturday, May 17, 2008, we repeatedly surveyed each host, initially running our script by hand and then, as we gained confidence, in a `cron` job. The result was a complete set of connection output for each host for each run. We did not attempt to restrict the host list to those hosts exhibiting weak certificates, thus avoiding the need to have a complete weak key list at the beginning of the survey — which was convenient since generating the key list is extremely time consuming.

The result of this process was a rather large data set — each day's data consumes approximately 200 MB and the entire set, representing connections to each server in our set on approximately a daily basis, is upwards of 30 GB

6.2 Generating Weak Keys

Analyzing the data requires identifying which servers were using weak keys, which in turn requires a list of weak keys. There is just 15 bits' worth of process ID entropy, but other parameters, described below, must also be accounted for. Previous weak-key generation efforts have used `getpid` interposition via `LD_PRELOAD`, but this approach does not scale. We instead created a patch to OpenSSL 0.9.8h that allows each of the relevant conditions to be simulated, and used the patched version to generate our corpus of weak keys.

Because the binary representation in memory of certain values is added to the entropy pool, not a canonical representation, our key generation must account for the target platform's endianness and native word size. In addition, the presence of a file called `.rnd` in the user's home directory affects the behavior of OpenSSL's command-line utilities. If it is present, its contents are added to the entropy pool. Accordingly, we must generate two sets of keys: ones assuming the presence of `.rnd`, one its absence. (Because of the Debian bug, the contents of the randomness file are not consulted; all 1024-byte files produce the same result.) When `.rnd` is missing, versions of OpenSSL before and after 0.9.8f have different behavior that we must again account for.[6] Debian-derived distributions shipped versions with both behaviors, so we must account for both.

[5]This took several iterations to get right. We had anticipated that OpenSSL would time out on its own, but this only applies to failed TCP connects. It will stall indefinitely once the TCP connect succeeded, which was an unpleasant surprise. This sort of stall happened often enough that we ended up having to parallelize our probes and use an alarm to kill stalled probes.

[6]Earlier versions will add their `struct stat` to the pool whether or not the `stat` call succeeds.

We generated keys for each of 32768 pids, on each of three platforms (little-endian 32-bit, big-endian 32-bit, and little-endian 64-bit), for each of three `.rnd` conditions (present; missing, old behavior; missing, new behavior). This is a total of 294,912 keys per key size.

We generated all 294,912 keys for every common keysize: 512, 768, 1024, 1536, 2048, 3072, 4096, and 8192 bits. In addition, we generated all 294,912 keys for various oddball keysizes we encountered in our survey, such as 1000 bits and 1023 bits. In the end, though, none of the odd-sized weak keys matched a certificate in our survey.

6.3 Processing the Data

To reduce the data (30+ GB for 173 days) to manageable form, we created a map from IP-time pairs to certificates. By analyzing each certificate once, we could determine most host properties we are concerned with[7] several orders of magnitude more quickly.

Once we know the status of each host on each day, we are still faced with the problem of how to analyze that data. A major complication is the natural turnover of certificates: we would expect to see hosts eventually get unaffected certificates even if administrators do not take explicit action to request a new certificate, due to the ordinary software upgrade and certificate expiry-and-replacement cycle: A new key generated on an upgraded server will automatically replace the old weak key. Even if the server has not upgraded (which Rescorla's results [16] suggest is often the case), the major CAs which dominate our survey attempted to detect vulnerable keys and would refuse to reissue the certificate. Thus, certificate expiration acts to force replacement of vulnerable keys; we would like to disentangle this effect from deliberate fixing.

Biology and epidemiology have developed an extensive array of *survival analysis* techniques designed to deal with situations like this where members of a population gradually undergoes a transition from one state to another (traditionally transitioning from alive to dead, hence the morbid name). The general idea is to look at the *hazard function* $h(t)$ which represents the probability of undergoing the transitional event at any period in time, and to compare the hazard functions between different groups. Techniques are also available for dealing with *censored* population members: those who disappear from view before undergoing the event. A good introduction to these techniques may be found in Kleinbaum [5]. We used Thierry Thernau's `survival` [17] package for R [14].[8]

One difficulty here is determining what we treat as an "individual" for the purpose of survival analysis, the host or the certificate. Unfortunately, neither is entirely satisfactory. We observed a number of cases of multiple machines exhibiting the same certificate (in the most extreme case, a single Akamai certificate appeared on 241 distinct hosts). In some cases, two hosts displaying the same certificate would fix on different days or one would disappear without being fixed. Thus treating machines as the basic unit is problematic.

We adopted the following strategy to deal with these inconsistencies: we grouped all hosts displaying the same initial certificate into a single unit, called a *host-cert* (HC), with an "event" assigned based on the final event observed from the HC: If A upgraded at time 1 and B stopped responding still unupgraded at time 2, we reported the event "Censored, 2"; on the other hand, if A upgraded at time 2 and the last contact with B was at time 1 but it was vulnerable at that time, we reported the event as "Fixed, 2". In general, groups of hosts with the same certificate behaved similarly, so other methodologies would likely have yielded similar results.

Like our initial decision of whether a host is vulnerable, our measurements of host certificate parameters (e.g., self-signed, key size, etc.) are based on our initial contact with a host. For instance, if a host had a vulnerable 1024-bit key, then transitioned to a vulnerable 2048-bit key, and then transitioned to a secure 1024-bit key, we would consider it to be a 1024-bit host.

In 22 cases, we saw hosts which had previously exhibited secure certificates suddenly start to display compromised certificates ("spontaneous generation"). We ignored these hosts.

For demographic information (e.g., cipher suite support), we identify which units (hosts or HCs) we are working in.

7. SURVEY RESULTS

Our main survey contacted 59100 hosts. Of those, 51838 answered at one time or another, with an average of 48555 hosts answering on any given day. During the course of the survey, we observed 751 vulnerable hosts (473 HCs). 507 of these hosts (241 HCs) were fixed during the survey period, with the remainder of the hosts either vulnerable on the final day ($n = 206$) or not responding ($n = 38$).

7.1 Demographics

Certificate Churn. Even in the absence of security vulnerabilities, CA-issued certificates must typically be reissued every 1 or 2 years. During the 173-day course of our study, 17579 (34%) of the hosts changed their certificates. As shown — for HCs — in Figure 4, vulnerable certificates are changed at a significantly different rate ($p < .001$; log-rank test) than other certificates. Qualitatively, this rate is much faster at the beginning of the survey period and then slows down and may be slightly slower (though we have no statistical tests confirming that it becomes slower) than the baseline out past 150 or so days.

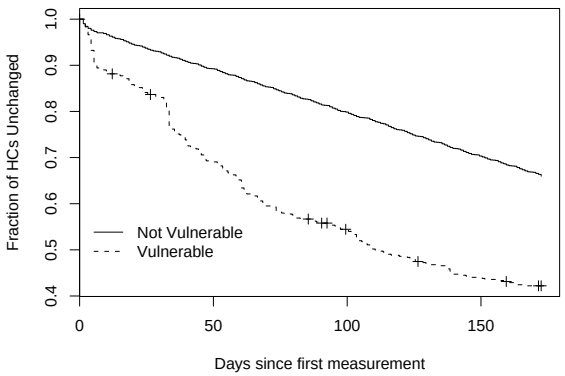

Figure 4: Rate of certificate churn

[7]Server cipher suite support is the notable exception.

[8]Note for non-statistician readers: different statistical packages often use subtly different algorithms even for well-understood operations. Thus, it is important for completeness to document the package used.

While the graph of churn of vulnerable certificates in Figure 4 and the graph of fixing of vulnerable certificates in Figure 1 are similar-looking, they represent different transitional events. If a host's certificate changes but the keys in both the old and new certificate are weak, this represents churn (so it is represented by a drop in Figure 4) but not fixing (so it is *not* represented by a drop in Figure 1). There were 34 such cases. In 16 of these cases, a certificate was renewed by the same CA and on the same weak key. In 8 more, certificates were issued by a new CA but on the same weak key. In 3 more, renewed certificates were issued by the same CA on a new key, with both the old and new keys weak. In 2 more, certificates were issued by a new CA and on a new weak key. In 2 cases, we saw several overlapping certificates on the same weak key. (In 4 of the cases above, we eventually saw a further new certificate with a good key.) In another 2 cases, self-signed certificates were updated with new, still-weak keys. In the last, spectacular case, a certificate was updated 55 times, with increasingly many CN fields giving a crude form of name-based virtual hosting; the first 10 certificates were all weak. We interpret the evidence above to mean that while the CAs do some checking (as discussed in Section 7.2), they either do not always check or they miss some weak keys.

Key Lengths. The vast majority of the HCs (approximately 93%) displayed 1024-bit RSA keys. The remainder of the were predominately 512-bit (2%) and 2048-bit (4%) RSA keys with a scattering of other key sizes. This is more or less as we expected: 1024 bits is the default key size output by most popular key-generation tools, such as the `mkcert.sh` tool included with Mod_SSL.

The distribution we observe of key sizes roughly agrees with the results of Lee et al.'s survey [9], which in the November 2006 survey reported 88% 1024-bit keys, 4% 512-bit keys, and 6% 2048-bit keys; both these surveys differ substantially from Murray's in 2000 [12], which found 70% 1024-bit keys, 23% 512-bit keys, and almost no 2048-bit keys.

Certificate Authorities. Only a small fraction of HCs had self-signed certificates (2%). Most HCs (93%) displayed certificates from CAs which had more than 100 certificates in our total sample.[9] This is strikingly different from reports by some other researchers. In particular, Netcraft [13] in January 2008 reported approximately one quarter of the certificates in their survey as being self-signed. We attribute this difference to our sampling methodology, which is biased towards servers which are heavily used, because it is those to which we will observe network traffic. As an additional datapoint, Murray gathered his list of servers by querying a search engine with various search terms and, in 2000, found less than 3% self-signed certificates.

To test this hypothesis, we portscanned randomly chosen machines on the Internet (using nmap's `-iR` flag). Several days of scanning recovered 20,214 hosts that accept connections on port 443; we then ran our survey tool against these, obtaining 19,299 certificates. Of these, 8,417 (44%) were self-signed. This seems to clearly indicate that sampling methodology matters — if we want to talk about SSL servers as a group we must first define what group we are interested in. In particular, it appears that commonly used (and hence "important") servers are more likely to have third-party certificates — thus allowing scalable authentication to arbitrary users — than would be suggested by simple random sampling.

Figure 5 shows the distribution of major certificate authorities by the number of HCs showing that certificate. The names here do not map 1-1 to X.509 issuerName values. Rather, we merged all CAs with the same brand name into a single bin: e.g., "VeriSign Class 3 Secure Server CA" and "VeriSign International Server CA — Class 3" are both in the "VeriSign" bin. This actually underestimates the influence of VeriSign, because VeriSign, Thawte, and Equifax are all owned by VeriSign and collectively they dominate the market. However, because each brand offers a somewhat different user experience, we have chosen to break out the CAs by brand. Note also that some of these CAs (e.g., Tor) have fewer than 100 certificates in a given day's data but have more than 100 certificates in aggregate.

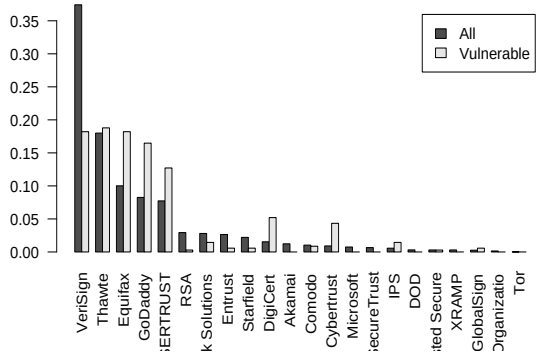

Figure 5: Distribution of CAs (day 1)

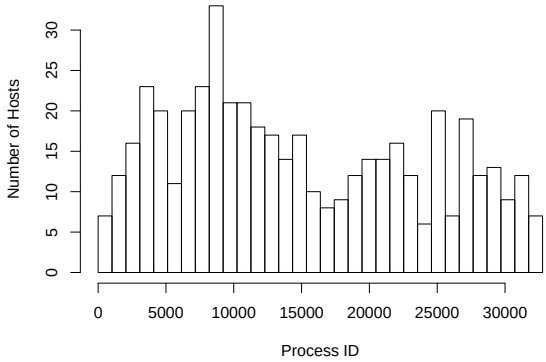

Figure 6: Distribution of Process IDs by cert

Because vulnerability in this case was out of the control of the user, we would expect the initial demographics of vulnerable certificates to be similar to those of non-vulnerable certificates. We indeed find this for the distribution of key lengths. But when we examine the distribution of CAs, as seen in Figure 5, we find that the distribution is quite different: VeriSign certificates are underrepresented in the population of vulnerable certificates. There are two plausible hypotheses about this difference: either VeriSign customers were less likely to use Debian and Debian-derived distributions, and were thus less likely to be affected, or VeriSign customers upgraded faster and so significantly more had upgraded by the time we started our survey. In Section 7.5 we discuss some evidence against the latter possibility.

[9]Note: to determine this 100 threshold we counted all certificates we saw, not just the first one for a given HC.

Cipher Suite Support. As discussed in Section 4.1, those servers that have weak long-lived keys but support DHE connections provide a higher degree of confidentiality against passive analysis than those with weak long-lived keys that support only RSA ciphersuites. Of the 746 hosts vulnerable on the first day of our study, 357 (48%) negotiated DHE with our OpenSSL client, indicating that they would likely negotiate DHE with a compatible browser. These comprise approximately one fifth of the market: Firefox will negotiate DHE with RSA certificates; Safari offers it, but below non-DHE suites that Mod_SSL supports; and Internet Explorer does not support `DHE_RSA` at all.[10] Compared to the vulnerable servers, a smaller fraction of all the hosts we surveyed on day one (30%) negotiated DHE with our client. By contrast, Lee et al. reported 58% penetration for `DHE_RSA`; we believe that some servers that support DHE nevertheless preferred another cipher suite from the list presented by our client, and that this partly explains the discrepancy. To verify this guess, we would have needed to make multiple connections to each server with different lists of supported cipher suites.

For the same reason, we do not have direct measurements for the level of support of symmetric algorithms. However, approximately 44% of the servers we surveyed on day one negotiated AES with our client, and more may support it, roughly consistent with Lee et al.'s report of 57% AES support. Amazingly, we found 18 hosts that negotiated an export cipher suite and 12 that negotiated single-DES.

Measuring Server Characteristics. Because the keys generated by OpenSSL depend on the state of the machine doing the key generation, we can remotely measure some properties of the server (or, more properly, the machine that generated the keys, though these are typically the same) that are ordinarily difficult to obtain. The first property of interest is the pid of the process that generated the key. We naively expected that the users would usually generate their keys shortly after boot, thus biasing the pid towards small numbers. However, while our data shows evidence of some biasing, the effect is not particularly strong, as shown in Figure 6, which displays the process ID histogram.

Some researchers have suggested that cloud-based systems are particularly vulnerable to such attacks because they start in a known state which is accessible to attackers [2]. Our findings have negative implications for the feasibility of such attacks: If keys are not usually generated soon after boot, the kernel will have a chance to gather additional entropy from interrupt timings.

We can also remotely determine the processor architecture. Approximately 80% of the HCs we observed were x86 32-bit and 20% were x86 64-bit with a trivial fraction being 32-bit big-endian machines.

7.2 Overall Upgrade Rate

Over the course of our survey 30% of the hosts (49% of HCs) initially exhibiting a vulnerable key remained vulnerable on the last day; the rest either transitioned to a non-vulnerable key or stopped responding. As shown — for HCs — in Figure 1, this was a relatively gradual process, with some notable discontinuities on days 2, 4, 5, 33, and 92. Examination reveals that the transitions on days 5, 33, and 92 were dominated by Equifax, USERTRUST and Thawte-issued certificates respectively. In the case of day 33, the hosts appear to be all operated by the same hosting provider; for day 91, the hosts appear to all be servers at the same site (`www09`, `www10`, etc.). There is no obvious pattern for days 2 and 4 and this may represent random chance, some as-yet undetermined action by the CAs, or (since these events took place about a week after the bug was disclosed) upgrading spurred by publicity for the bug.

As is apparent from Figure 1, we see a quite different pattern of fixing from that reported by either Ramos et al. [15] or Rescorla [16]. Instead of a fast exponential decay followed by almost no change, we see a very gentle curve with substantial rates of fixing out to 5–6 months. One natural explanation for the curve's shape is the significant baseline hazard of certificate expiration. We would expect this hazard function to be constant because the certificate expiration date is roughly randomly distributed. The roughly straight "not vulnerable" line in Figure 4 provides some support for this belief.

While it seems likely that the baseline certificate expiration rate is a major factor in the long upgrade curve, we suspect that other factors are relevant as well. As a heuristic test, we removed all certificates whose expiration time was less than 30 days from the upgrade time and plotted a new survival curve. This still exhibits a substantial amount of upgrading past 75 days. An alternative way to examine this question is to compute the *hazard ratio*[11]: $h(t)/h_0(t) = e^{\beta(t)}$ for certificates which are vulnerable versus those that are not. As shown in Figure 7, which displays a spline fit of $\beta(t)$ (the dashed lines indicate two standard error confidence bounds), the excess upgrading does not smoothly decrease, as we expect from previous work, but rather increases up to about day 45 and then decays afterwards. We note that an eventual slowdown below the baseline rate is not unexpected: If certificates are changed ahead of schedule, then we don't see the corresponding scheduled replacements.

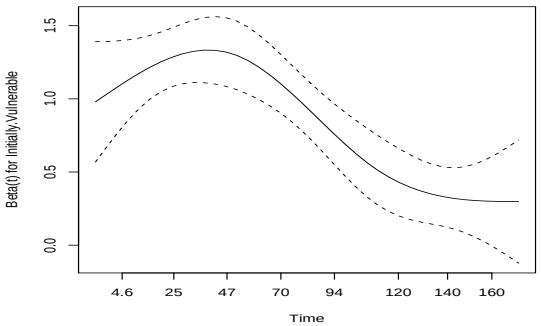

Figure 7: Log hazard ratio for vulnerable versus non-vulnerable certificates

Another potential contributor to this much longer than expected updating is the unique nature of this vulnerability. With ordinary vulnerabilities, the perceived (and according to Ramos, actual) risk of compromise is very high imme-

[10] More specifically, Internet Explorer running on Windows 2000, Windows XP, and Windows Server 2003 will negotiate DHE with DSS certificates, which are not deployed for Internet-wide HTTPS; Internet Explorer running on Windows Vista and Windows Server 2008 will in addition negotiate ECDHE with RSA certificates, but the version of OpenSSL deployed on Debian-derived distributions ships without any elliptic curve support.

[11] It is conventional to work in log units here.

diately after announcement and is a step function — either your machine is compromised or it is not. By contrast, the risk of having a weak RSA key due to passive attack is linear in the number of days it is used and, as indicated in Section 4.2, upgrading the server certificate has only a very small effect on active attack because most clients do not check CRLs. Thus, it may be rational for an administrator to delay upgrading their system and certificates for longer than they would for an ordinary vulnerability.

7.3 Factors Affecting the Upgrade Rate

In addition to the gross upgrade rate, an important question to ask is: Are there factors that predict whether or when a vulnerable certificate/host will be upgraded? To compare the hazard functions, we used the popular Cox Proportional Hazards model, which assumed that the hazard functions $h(t)$ for each combination of predictors are roughly constant for all values of time and attempts to compute the hazard ratio.[12] The advantage of the Cox model is that it is *non-parametric*, i.e., it does not require an underlying analytic model for the hazard function and it gives a single numeric result for the increased risk. The disadvantage is that it does not give a meaningful numeric result when the hazard ratio is not constant (the "proportional hazards assumption").

We considered a large number of candidate predictors and fit the Cox proportional hazards model using the R `coxph` function and the `stepAIC` [19] procedure for automatic model selection. Due to the small size of the data set, we did not consider interactions of predictors because many interactions had zero counts. Chi-squared tests were consistent with the proportional hazards assumption for all covariates but key size and expiry during the study so we stratified on those variables. Inspection shows some but not undue evidence of time dependence of $\beta(t)$ in the others. The results are shown at the end of this section in Figure 12.

This procedure resulted in four predictors with potentially significant effects: key size, expiry during the study, CA type, and the number of hosts displaying a particular cert. We discuss these predictors below.

Key Size. Figure 8 shows the rate of upgrading stratified by key size. There are not enough 512- and 4096-bit keys to draw any conclusions from, but 2048-bit keys are clearly upgraded much faster than 1024-bit keys ($p < .01$; log-rank test) This, too, is unsurprising; 1024-bit keys are standard practice and only the extremely paranoid use 2048-bit keys. We would expect those users to be more attentive to security and more responsive to security issues of this type. Because this covariate exhibits time-dependence, we do not have an estimate of the hazard ratio.

Expiry During the Study. Figure 9 shows the rate of upgrading stratified by whether the certificate expires before the end of the study (including before the start of the study). Interestingly, these HCs fix slower initially but then faster after 100 days or so. This is roughly consistent with Figure 7, which shows a similar fast pattern followed by slowdown. Because this covariate exhibits extreme time-dependence, it does not make sense to estimate the hazard ratio.

CA Type. Figure 10 shows the rate of upgrading stratified by the type of issuer, "big CA" (> 100 total certificates in our sample) "small CA" (between 2 and 99 certificates), "Other" (1 certificate) or "self-signed". Qualitatively, hosts with self-signed certificates are significantly ($p = .01$) slower to fix than those with non-self-signed certificates. This isn't an unexpected result, as low-value and test servers often use self-signed certificates, whereas serious commercial organizations generally need to run certificates from third party CAs. There is no statistically significant difference between the other categories, although given the small number of certificates in the "Other" category, this is potentially an issue of insufficient statistical power.

Certificate Instances. As discussed in Section 6.3, we saw a number of instances of the same certificate appearing on multiple IP addresses. Because large services often

[12] More formally, for each predictor i, we fit a coefficient β_i. If $\mathbf{X}$ denotes the presence or absence of each predictor, then we have $h(t, \mathbf{X}) = h_0(t)e^{\sum_{i=0}^{p} \beta_i X_i}$.

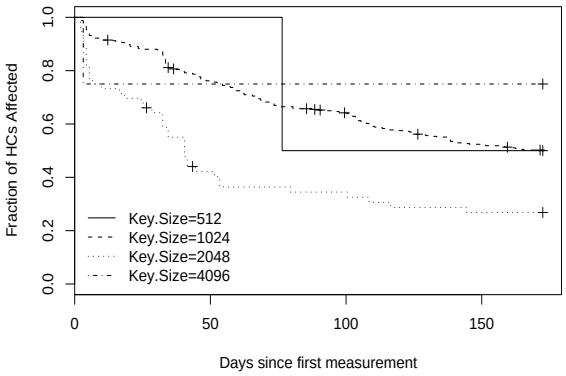

Figure 8: Upgrading rate by key size

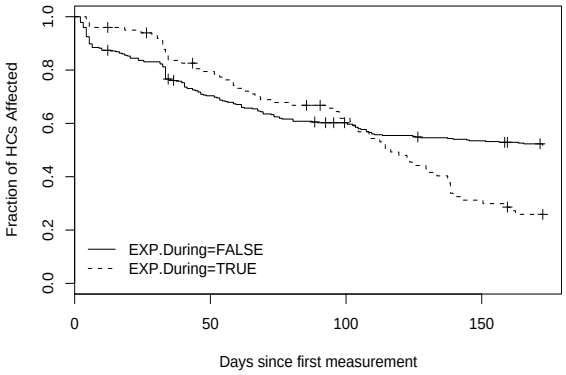

Figure 9: Upgrading rate by expiry time

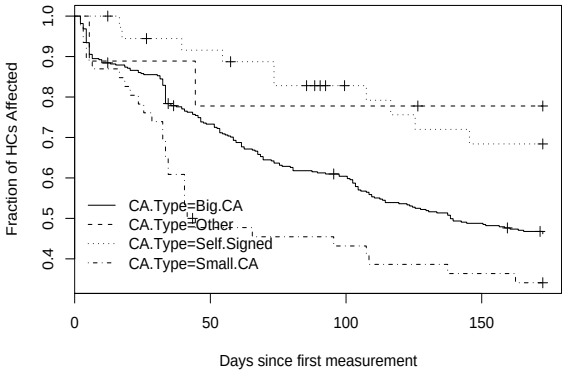

Figure 10: Upgrading rate by CA type

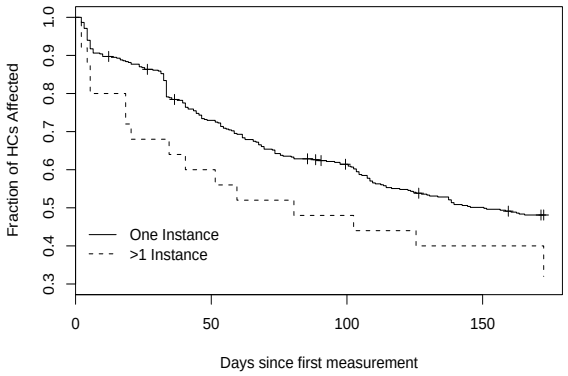

Figure 11: Upgrading rate by certificate instances

	coef	exp(coef)	se(coef)	z	p
CA.TypeOther	-1.12	0.33	0.75	-1.49	0.13
CA.TypeSelf.Signed	-0.89	0.41	0.33	-2.70	0.01
CA.TypeSmall.CA	0.14	1.15	0.22	0.63	0.53
Mult.HostsTRUE	0.57	1.77	0.26	2.23	0.03

Figure 12: Predictors of upgrading rate

operate multiple servers, this can be used as an (imperfect) proxy for size. As Figure 11 shows, certificates that appear on multiple hosts seem to be fixed faster. We have not investigated whether there is a dependency on the number of copies beyond this.

Other Factors. Even if there are no between-group differences between CAs based on size, it's possible that there are within-group differences, for instance due to CA customer population or notification strategies. Exploratory data analysis using both CA names and the binning strategy of Figure 5 doesn't show any statistically significant results and given the large number of factors and the concomitant risk of data mining combined with the messiness of the data when viewed qualitatively, we cannot say with confidence that customers of one CA upgrade faster than others. We also examined architecture, which did not have a significant effect after controlling for other significant factors.

7.4 Sources of Error

Our data is subject to a number of sources of error. Most importantly, because we are identifying hosts by IP address rather than hostname, the data in this survey is affected by renumbering, which may cause loss of contact with hosts or host substitution. However, this can generally be detected by examining the certificates, and as Figure 4 shows, we see a fairly high degree of host stability.

The list of servers we surveyed may not be without bias. It consists of the servers contacted, over approximately two-month period, by users of UC San Diego's campus network, a large and diverse population of faculty, staff, and students including UCSD's residential undergraduate colleges. It may, nevertheless, display some US-centrism or be skewed towards those sites interesting to members of an academic community.

Additionally, it is possible that due to limitations in our key-generation code, we are missing some vulnerable keys. First, because we did not have access to a 64-bit big-endian machine running Debian, we were not able to check that we can correctly generate keys for that platform; we believe that Debian is not widely deployed on these machines. Second, although we attempted to find all possible factors affecting key generation, it is possible that there are other variations we missed. Users who use a non-default generation method, either applying different command-line options to `openssl genrsa` or bypassing that program altogether, will obtain keys different to those we generated.

While the results of our survey are quite detailed, in retrospect it would have been useful to capture additional data. In particular, had we used the `-debug` flag to OpenSSL's `s_client`, we could have captured the entire handshake including the ServerHello, which would give us the ServerRandom. Fortuitously, however, because the SSL session identifier is randomly generated and is part of the output, we still have a marker for the state of the server PRNG.

It would also have been nice to gather more information about the server. For instance, we could have probed it to determine which cipher suites it would accept. The current data indicates only which suite was chosen, which is likely to be just one of many it accepts. Those servers that agreed to negotiate a `DHE_RSA` cipher suite with our survey client would also negotiate a `DHE_RSA` with those browsers that support it, which make up approximately one fifth of the market, as we discuss in Section 7.1. In addition, because we hang up after the SSL handshake, we do not gather any HTTP-level information about the server. Although server version strings are not particularly reliable, this might be useful to have in the future.

7.5 Extrapolating The Missing Data

As mentioned in Section 1, the Debian OpenSSL vulnerability was announced 4–5 days prior to our first survey. This creates a potential source of error because we cannot directly measure upgrading of hosts that occurred before our study period. To try to get a handle on this, we looked at certificates issued in the period between the announcement and our first survey. We were not able to obtain information for all certificates, but VeriSign kindly provided us with the fate of the *predecessor* for each VeriSign and RSA branded certificate in our data set with an issuance date within a day or two of our study period ($n = 366$). Of those, less than 10% were revoked and only 3 of the revoked keys were weak. We have not checked the keys for the unrevoked certificates; it is possible that some of them are weak and so this is an underestimate. However, most of the changed certificates were marked either as normally reissued or as a first certificate in the system. Thus we believe that, for VeriSign at least, there was not a large amount of fixing prior to our study. We hope to expand the scope of this analysis in collaboration with VeriSign and other CAs.

8. CONCLUSIONS

Much is known about how users and administrators respond to software vulnerabilities, but comparatively little is known about how they respond to cryptographic compromise. Although anecdotal reports indicate that even when keys are known to be compromised administrators will not change them, there is no readily available empirical evidence. The Debian OpenSSL vulnerability provided us with a unique ability to remotely measure administrator response to key compromise.

Using a survey of more than 50,000 SSL/TLS-enabled Web servers, we have examined the rate of certificate upgrading. We find that unlike other vulnerabilities which

have been studied and typically show a short, fast, fixing phase followed by levelling off, certificates were replaced on a slower cycle with substantial fixing extending well past five months after the announcement. We also find that in some cases certificate authorities continued to issue certificates to weak keys long after the vulnerability was announce. In the process of this research we have developed extensive tooling and a new SSL survey data set that that allows us to re-examine existing work on SSL server demographics. Our dataset is available to other researchers interested in studying questions of real-world SSL deployments.

Acknowledgements

We are grateful to Dan Boneh for discussions on how to attack DHE and to Nagendra Modadugu for advice on methodology. Josh Benaloh and David Pickett provided key details of the behavior of Internet Explorer. Florian Weimer and Dirk-Willem van Gulik helped identify the variations possible in key generation. Patrick Nehls and the UC San Diego security office provided the original list of hosts without which this study would not have been possible. Thanks to Terry Therneau for assistance with the Cox proportional hazards analysis and for providing the `survival` package. This document was prepared using Sweave [10]. We would especially like to thank Jeff Barto and Rick Andrews for their assistance with obtaining data on VeriSign's certificates and notification policy.

We also benefited from discussions with Jennifer Granick, Joe Hall, Candice Hoke, Jim Hughes, Cullen Jennings, Moni Naor, and Melanie Schoenberg.

Finally, we are grateful for the comments and suggestions of the anonymous IMC reviewers.

This material is based upon work supported in part by the National Science Foundation under Grants No. 0831532 and 0831536 and supported in part by a MURI grant administered by the Air Force Office of Scientific Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Air Force Office of Scientific Research.

9. REFERENCES

[1] P. Abeni, L. Bello, and M. Bertacchini. Exploiting DSA-1571: How to break PFS in SSL with EDH, July 2008. `http://www.lucianobello.com.ar/exploiting_DSA-1571/index.html`.

[2] A. Becherer, A. Stamos, and N. Wilcox. Cloud computing security: Raining on the trendy new parade. Presented at BlackHat USA 2009, July 2009. Online: `http://www.isecpartners.com/files/Cloud.BlackHat2009-iSEC.pdf`.

[3] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, pages 66–70, Jan. 1996.

[4] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, Internet Engineering Task Force, Dec. 2005.

[5] D. G. Kleinbaum. *Survival Analysis: A Self-Learning Text*. Springer, 1996.

[6] B. Laurie. Debian and OpenSSL: The aftermath, May 2008. `http://www.links.org/?p=328`.

[7] B. Laurie. Vendors are bad for security, May 2008. `http://www.links.org/?p=327`.

[8] B. Laurie and R. Clayton. OpenID/Debian PRNG/DNS cache poisoning advisory, Aug. 2008. `www.links.org/files/openid-advisory.txt`.

[9] H. Lee, T. Malkin, and E. Nahum. Cryptographic strength of SSL/TLS servers: Current and recent practices. In C. Dovrolis and M. Roughan, editors, *Proceedings of IMC 2007*, pages 83–92. ACM Press, Oct. 2007.

[10] F. Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–80. Physica Verlag, Heidelberg, 2002.

[11] M. Mueller. Debian OpenSSL predictable PRNG bruteforce SSH exploit, May 2008. `http://milw0rm.com/exploits/5622`.

[12] E. Murray. SSL server security survey, July 2000. Archived copy online: `http://web.archive.org/web/20031005013455/http://www.lne.com/ericm/papers/ssl_servers.html`.

[13] Netcraft. Netcraft SSL survey. `news.netcraft.com/SSL-Survey/`, Jan. 2008.

[14] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[15] T. Ramos. The Laws of Vulnerabilities. RSA Conference, 2006. `http://www.qualys.com/docs/Laws-Presentation.pdf`.

[16] E. Rescorla. Security holes... who cares? In V. Paxson, editor, *Proc. 12th USENIX Security Symp.*, pages 75–90. USENIX, Aug. 2003.

[17] S original by Terry Therneau, ported by Thomas Lumley. *survival: Survival Analysis, including Penalised Likelihood*. R package version 2.34.

[18] The Debian Project. openssl – predictable random number generator. DSA-1571-1, May 2008. `http://www.debian.org/security/2008/dsa-1571`.

[19] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002.

[20] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, Internet Engineering Task Force, Jan. 2006.

APPENDIX

A. ATTACKING APACHE WITH DHE

In this appendix, we show how an attacker who observes every connection to an Apache SSL server can track the entropy pools of the parent process and its children. This allows the attacker to decrypt all traffic in DHE sessions the server negotiates. (Ordinary RSA sessions remain secure.) Only the entropy used in handling connections needs to be weak for this attack to succeed; the server's long-lived key can be strong.

Compared to simple process-per-child servers, Apache introduces a number of complicating factors. First, a single worker process will handle multiple connections in sequence. Each connection will call `RAND_bytes` one or more times, mixing the entropy pool. Even with the Debian bug,

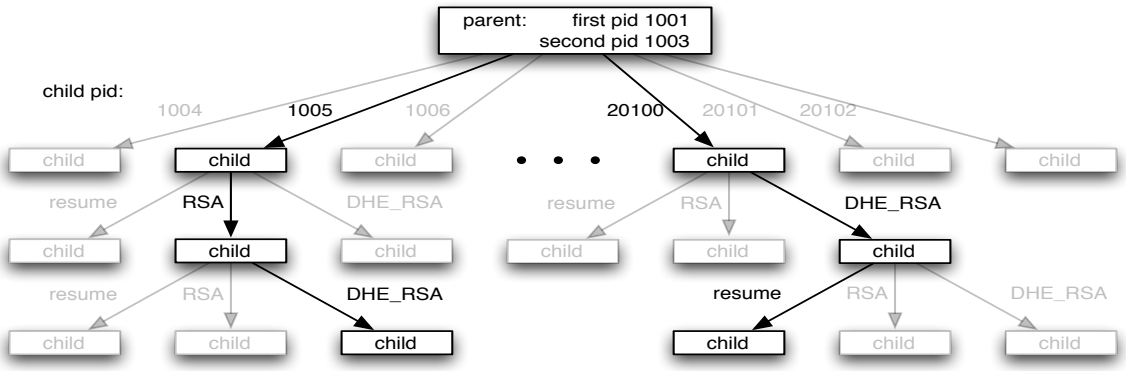

Figure 13: Apache child entropy-pool state search tree.

then, the random values obtained by the process for the first connection it handles will be different from those obtained for the second and subsequent connections. Nor can an attacker compute the random values for the first k connections for each pid, obtaining a table of size $32768 \times k$: the pattern of `RAND_bytes` calls made by the process in handling a connection is different depending on whether the connection requires a full handshake or just a session resume and whether the cipher suite negotiated in the handshake is `RSA` or `DHE_RSA`. Starting from some initial known entropy pool, one obtains a ternary tree; the cost of precomputation is exponential in the depth. Second, initialization, including the generation of cryptographic parameters, is carried out in a parent process that forks off child processes to handle connections. The entropy pool of a child process has both parent and child pids mixed in, so an attacker may have to precompute each of $2^{15} \cdot 2^{15}$ values — though for lightly loaded servers the child pids are likely to be near the parent pids.[13] Third, because Apache employs a "thundering herd" architecture, an attacker will not *a priori* know which child process will handle a particular connection.

Attacks against Apache servers are still possible, but they require the attacker to observe and record *all* traffic to the server from the moment it starts accepting connections.[14] Prior to the attack, the attacker precomputes the entropy pool for each possible first parent, second parent, and child pid, along with the ServerRandom that would be sent by the first connection; we discuss the computation and storage this requires in Appendix A.3 below. Now the attacker processes the first connection to the server. The ServerRandom will be one of the values listed in the table. The attacker has identified the parent pids and the pid of one of the children. He will now track the contents of the entropy pool for the child process he has discovered. Subsequent connections will either be first connections to other children, which the attacker will look up in his table, or second or later connections to already-known children, for which the attacker will appropriately update his local copy of that child's entropy pool. Crucially, all the values that affect the pattern of `RAND_bytes` calls made by the server process are transmitted in the clear as part of the SSL handshake. Figure 13 shows the evolution of the attacker's knowledge.

We have implemented our attack on Ubuntu 7.10 (Gutsy) running the default Apache2 MPM-Prefork package with OpenSSL version 0.9.8e-5ubuntu3, which contains the Debian SSL vulnerability. Apache2 MPM-prefork, like Apache 1.3, uses processes instead of threads to handle requests. While the tables we construct are specific to this version of Apache and its default Ubuntu configuration, additional tables could be easily built for other common configurations. The remainder of this section describes the details of our attack.

A.1 Building ServerRandom Tables.

First, we created lookup tables to allow us to determine, given a session ID or ServerRandom value, the pids of the Apache parents and worker child. We constructed a simulator for Mod_SSL that takes as input three pids pid_1, pid_2, and pid_3, and executes the exact same calls to the OpenSSL PRNG that Apache makes when initializing the server with parent pids pid_1 and pid_2 and worker pid pid_3. Once the simulated worker process requests 28 bytes for the ServerRandom (after requesting 32 bytes for the session ID), we record this value and the three pids in our table.

A second table is required to handle the case where the first connection to a new child is a session resume, since in this case the ServerRandom is the first value drawn from the PRNG, not the second after the session ID. It is possible to combine both tables by recording both the session ID and ServerRandom in a single table; in a session resume, the first 28 bytes of the ServerRandom will be a prefix of what would otherwise be the session ID.

A.2 Compromising DHE Sessions.

To track each child's entropy pool as it evolves over multiple connections, we modify the simulator we used to generate the ServerRandom table so that simulates connections of specified types in the sequence after generating the initial ServerRandom. Because the number of `RAND_bytes` calls

[13] In fact, in an additional perversity, initialization is carried out in each of *two* parent processes as Apache forks away from its controlling terminal and session. This does not mean that the attacker must precompute $(2^{15})^3$ values, however: The second process is created soon after the first, and Debian's kernel assigns pids sequentially.

[14] An attacker who can use a DoS bug to crash child processes will have a somewhat easier time mounting the attack, because it is the child processes whose entropy evolves over time: the attacker will crash each child process, causing the parent to fork fresh new children; connections from that point on will be vulnerable even if the attacker did not observe earlier connections.

for blinding operations depends on the server's public key, this simulator must be tailored to the specific server we are attacking.

We use this simulator as follows. We use `ssldump`[15] to obtain a parsed version of all of the connections to the server. We extract the connection type and other values relevant to our analysis from the dump using a Perl script. As we step through the history of connections to the server, we keep track of the worker processes we have identified as being active, and the sequence of connection types each has encountered thus far.

For each new connection that we examine, we first check if it is handled by a worker process that we already know about. To do this, we use our simulator to increment the state of each known worker process and then check if the ServerRandom emitted matches the ServerRandom of the new connection we are examining. If there is a match, we have determined that the connection is handled by an existing worker process that we are already tracking. If not, the connection must have been handled by a new worker process; we can use our table of ServerRandom values to find the new process' pid and start tracking its state.

As we sequentially examine sessions in the ssldump, we can completely determine the PRNG state of every worker process at each point, so for the DHE sessions we can determine the ephemeral Diffie-Hellman private key, compromising those sessions.

A.3 Resources Required for the Attack.

Our attack platform was a machine with a single 3.2 GHz Intel Pentium 4 with 1 GB RAM. We performed a proof-of-concept attack under controlled conditions, building a subtable of the full lookup table discussed above, with first parent pid in the range $[5000, 5835]$; the second server pid either the next or next-but-one; and worker pids up to one hundred after the second server pid. On our lightly-loaded test machine, rebooted before each trial, this small table always sufficed.

The table, containing approximately 160,000 entries, is 10 MB. We estimate that the full table of 2^{35} entries[16] would take less than 4 TB.

Generating our small lookup table by naïvely simulating each pid triple took 15 hours. Using this approach to generate the full table would take several million hours of computer time. However, checkpointing the computation after the parent initialization has completed can cut this time down by several orders of magnitude. At 2^{20} cost, this gives a lookup table of entropy pool states for each pair (pid_1, pid_2) of parent pids. One then uses the appropriate checkpoint entry to derive the entropy pool of each child pid pid_3. This gives all (pid_1, pid_2, pid_3) entries in the table without needlessly repeating the most costly part of the computation. Our experiments show that the two parent initializations, which require RSA keypair generation and make hundreds of PRNG calls, take 135 ms on average for each parent, whereas the final worker process stage is much faster — about 0.04 ms. This suggests a total cost for computing the table by this method of $\left(2^{20} \times 2 \times 135 \text{ ms}\right) + \left(2^{35} \times 0.04 \text{ ms}\right) \approx$ 460 hr, though the actual cost may be somewhat higher due to IO overhead.

The checkpointed workload, like the naïve one, is highly parallelizable. The basic subtask is to compute the table entries for a particular parent pid pair (pid_1, pid_2) and all possible child pids. This subtask takes less than 1.6 s and is totally independent of any other subtask. It would take a cluster of 20 machines just a day to compute the entire table.

[15] `http://www.rtfm.com/ssldump/`.

[16] This is any of 2^{15} values for the initial server pid and child pid, and a smaller 2^5 range for the second server pid.