UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Systematic Analysis of the Juniper Dual EC Incident

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Jacob Maskiewicz

2016

DEDICATION

Many thanks to my wonderful bride Lily, for her loving support and superb editing skills. I wouldn't sound nearly as smart without your help.

Thanks to my parents for their support, and to my mom who showed me that there's always more to learn, and inspired me to do my own research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Chapters 1 to 3 are a reprint of material submitted to Usenix Security 2016 for publication. Research was a collaboration with Stephen Checkoway, Shaanan Cohney, Christina Garman, Matthew Green, Nadia Heninger, Eric Rescorla, Hovav Shacham, and Ralf-Philipp Weinmann. The thesis author was a primary researcher of this work.

ABSTRACT OF THE THESIS

A Systematic Analysis of the Juniper Dual EC Incident

by

Jacob Maskiewicz

Master of Science in Computer Science

University of California, San Diego, 2016

Professor Hovav Shacham, Chair

In December 2015, Juniper Networks announced that unknown attackers had added unauthorized code to ScreenOS, the operating system for their NetScreen VPN routers. This code created two vulnerabilities: an authentication bypass that enabled remote administrative access, and a vulnerability that allowed passive decryption of VPN traffic. Reverse engineering of ScreenOS revealed that the first of these was a conventional back door in the SSH password checker. The second is far more intriguing: a change to the Q parameter used by the Dual EC pseudorandom number generator. It is widely known [7, 33] that Dual EC has the unfortunate property that an attacker who chooses Q can, from a small sample of the generator's output, predict all future outputs. In a 2013 public statement, Juniper noted the use of Dual EC but claimed that ScreenOS included countermeasures against this form of attack.

In this work, we report the results of an independent analysis of the ScreenOS randomness subsystem, and its interaction with the IKE VPN key establishment protocol. Due to apparent flaws in the code, Juniper's countermeasures against a Dual EC attack are never executed. Moreover, by comparing sequential versions of ScreenOS, we identify a cluster of additional changes that were introduced concurrently with the inclusion of Dual EC in a single 2008 release. Taken as a whole, these changes render the ScreenOS system vulnerable to passive exploitation by an attacker who selects Q. We demonstrate this by installing our own parameters, and showing that it is possible to decrypt VPN traffic given a passive capture of a single IKE handshake.

Chapter 1

Background

1.1 Introduction

Random number generation is critical to the implementation of cryptographic systems. Random numbers are used for a variety of purposes, including generation of nonces and cryptographic keys. Because generating a sufficient quantity of true random numbers via physical means is inconvenient, cryptographic systems typically include deterministic *pseudorandom number generators* (PRNGs) which expand a small amount of secret internal state into a stream of values which are intended to be indistinguishable from true randomness.

Historically, random number generators have been a major source of vulnerabilities [6, 13, 16, 22, 36]. This is because an attacker who is able to predict the output of a PRNG will often be able to break any protocol implementation dependent on it. For instance, they may be able to predict any cryptographic keys (which should remain secret) or nonces (which should often remain unpredictable). Past PRNG failures have resulted from a failure to seed with sufficiently random data [13, 16] or from algorithms which are not *secure*, in the sense that they allow attackers to recover the internal state of the algorithm from some public output.

A number of Juniper NetScreen-branded VPN/Firewalls use the NSA-designed Dual EC PRNG [4, 21]. Dual EC has the problematic property that an attacker who knows the discrete logarithm of one of the input parameters (Q) with respect to a generator point, and is able to observe a small number of consecutive bytes from the PRNG, can then compute the internal state of the generator and thus predict all future output. In December of 2015, Juniper reported [17] that at some point in 2012, an attacker had modified the code in their version control system to substitute an alternate value of Q in place of the initial Q value generated by Juniper. In this paper, we report on the impact of this change — based on extensive reverse engineering and experimental testing — and of the broader security properties of the Juniper NetScreen PRNG design.

Summary of our findings. Our analysis shows that that the current Juniper ScreenOS PRNG implementation is vulnerable to efficient state recovery attacks conducted by an attacker who selects the Q value. Surprisingly, this finding is not an inevitable result of the known attacks on Dual EC, but instead stems from a collection of design choices made by Juniper in 2008.

Between ScreenOS 6.1 and 6.2.0r1, we identified a constellation of changes made to both the PRNG and IKE implementations that substantially predispose the IKE/IPSec implementation to state recovery attacks on the Dual EC generator. These changes, which were introduced concurrently with the addition of Dual EC, create a "perfect storm" of vulnerabilities that combine to produce a highly effective *single-handshake* exploit against the ScreenOS IKE implementation. Moreover, we identify several implementation decisions that superficially appear to reduce exploitability, but that on closer examination actually facilitate the attack.

To validate the accuracy of our findings, we implement a proof of concept exploit against a ScreenOS 6.2 device and show that when the device is configured with a Q parameter of our choosing, our attacks can efficiently decrypt VPN connections from a single handshake, without seeing any other traffic. Moreover, we discuss the impact of different IPSec versions and configurations on the attack, and show that configuration decisions can substantially affect the exploitability of the device, in some cases rendering the device entirely secure.

Outline of the paper. The remainder of this paper is structured as follows. In Section 1.2 we provide background on the Dual EC PRNG. In Section 1.3 we discuss Juniper's vulnerability disclosure and the research questions it raises. In Sections 2.1 and 2.2 we describe the details of the ScreenOS PRNG and its interaction with the IKE key exchange protocol. In Section 2.3 we describe a practical exploit of the NetScreen IPsec functionality under the assumption that the attacker knows the discrete log of Q. In Section 2.4, we discuss how one can remotely detect vulnerable ScreenOS versions. In Section 3.1 we discuss the broader implications of this issue.

1.2 Dual EC Background

In this section, we describe the Dual EC pseudorandom number generator and the attack on it described by Shumow and Ferguson [33], with some details on how ScreenOS implements Dual EC.

Dual EC comes in a variety of forms which affect the difficulty of the Shumow–Ferguson attack. There are two slightly different NIST standards for Dual EC, which also contain optional features. There are three standard elliptic curves which can be used, and implementors are free to make a number of software engineering choices. For concreteness, we describe Dual EC as implemented in Juniper's ScreenOS below. For more details on other forms of Dual EC, see Checkoway et al. [7].

Dual EC has three public parameters: the elliptic curve, and two distinct points on the curve called P and Q. ScreenOS uses the elliptic curve P-256 and sets P to be P-256's standard generator as specified in NIST Special Publication 800-90A [28]. That standard also specifies the Q to use, but ScreenOS uses Juniper's own elliptic curve point instead. The finite field over which P-256 is defined has roughly 2^{256} elements so points on P-256 consist of pairs of 256-bit numbers (x, y) that satisfy the elliptic curve equation. The internal state of Dual EC is a single 256-bit number s.

In ScreenOS, Dual EC is always used to generate 32 bytes of output at a time. Let $x(\cdot)$ be the function that returns the x-coordinate of an elliptic curve point; $\|$ be concatenation; $lsb_n(\cdot)$ be the function that returns the least-significant n bytes of its input in big-endian order; and $msb_n(\cdot)$ be the function that returns the most-significant n bytes. Starting with an initial state s_0, Dual EC generates 32 pseudorandom bytes *output* and a new state s_2 as follows,

$$\begin{aligned} s_1 &= x(s_0P) \\ r_1 &= x(s_1Q) \\ s_2 &= x(s_1P) \\ r_2 &= x(s_2Q) \\ \mathit{output} &= \mathit{lsb}_{30}(r_1) \parallel \mathit{msb}_2\big(\mathit{lsb}_{30}(r_2)\big), \end{aligned}$$

where sP and sQ denote scalar multiplication.

In 2007, Shumow and Ferguson [33] noted that if the discrete logarithm $e = \log_P Q$ (i.e., the integer e such that $eP = Q$) were known, then seeing *output* would reveal the Dual EC internal state. The key insight is that one can obtain $d = \log_Q P = e^{-1} \bmod n$, where n is the group order, and then multiplying the point s_1Q by d yields the internal state $x(d \cdot s_1Q) = x(s_1P) = s_2$. Although s_1Q is itself not known, 30 of the 32 bytes of its x-coordinate (namely r_1) is the first 30-bytes of *output*.

This insight gives rise to the simple procedure to recover s_2. For each of the 2^{16} 256-bit integers r such that $\mathit{lsb}_{30}(r)$ equals the first 30-bytes of *output*, check if r is a valid x-coordinate of a point on the curve.[1] In other words, find a point R such that $x(R) = r$. Roughly half of the r values will be valid x-coordinates.[2] For each such R, compute $s' = x(dR)$ and $r' = x(s'Q)$. If the correct $r = r_1$ is chosen, $\mathit{msb}_2\big(\mathit{lsb}_{30}(r')\big)$ will be equal to the last two bytes of *output* and $s' = s_2$, the new internal state.

The one complication with the above procedure is that there may be several

[1]This procedure is sometimes called point decompression and involves computing a modular square root.

[2]Each r that is an x-coordinate of some point R is also an x-coordinate of the point $-R$. It doesn't matter which point is chosen as R and $-R$ differ only in the "sign" of their y-component.

values of r such that $msb_2\big(lsb_{30}(r')\big) = lsb_2(output)$ and each such r corresponds to a potential internal state s'. In practice, this is a minor complication as it's exceedingly rare for there to be more than three such r.

1.3 History of the Juniper Incident

After NIST recommended against the use of Dual EC [28] in response to post-Snowden concerns about the default value of Q, Juniper published a knowledge base article [19] explaining their use of Dual EC in ScreenOS, the operating system powering its NetScreen firewall appliances, stating that although those products used Dual EC:

> ScreenOS does make use of the Dual_EC_DRBG standard, but is designed to not use Dual_EC_DBRG as its primary random number generator. ScreenOS uses it in a way that should not be vulnerable to the possible issue that has been brought to light. Instead of using the NIST recommended curve points it uses self-generated basis points and then takes the output as an input to FIPS/ANSI X.9.31 (sic) PRNG, which is the random number generator used in ScreenOS cryptographic operations.

The first of these countermeasures — self-generated basis points[3] — is not completely satisfactory because it depends on Juniper generating Q in such a way that nobody knows its discrete log, which they have not verifiably demonstrated. However, the second countermeasure — if implemented correctly — defends against the current publicly-known attacks on Dual EC because those attacks rely on having Dual EC output rather than a one-way function of that output, so even an attacker who knew the discrete log of Q would be unable to recover the PRNG state.

This was the situation in December 17, 2015 when Juniper issued an out-of-cycle security bulletin [17] for two security issues in ScreenOS:

- CVE-2015-7755[4](“Administrative Access”)
- CVE-2015-7756[5](“VPN Decryption”)

[3]While the Juniper article says “points”, actually only Q differs from the NIST default values. Juniper’s implementation uses the default P value

[4]`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7755`

[5]`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7756`

This announcement was particularly interesting because it was not the usual report of developer error, but rather of malicious code which had been inserted into ScreenOS by an unknown attacker:

> During a recent internal code review, Juniper discovered unauthorized code in ScreenOS that could allow a knowledgeable attacker to gain administrative access to NetScreen® devices and to decrypt VPN connections. Once we identified these vulnerabilities, we launched an investigation into the matter, and worked to develop and issue patched releases for the latest versions of ScreenOS.

The "Administrative Access" vulnerability was determined to be a back door in the SSH [37] daemon that would have allowed anyone who knew the correct password to log in with administrative access. This issue has been extensively discussed by Moore [27]. The second issue, however, turns out to be far more technically interesting. According to Juniper's advisory:

> VPN Decryption (CVE-2015-7756) may allow a knowledgeable attacker who can monitor VPN traffic to decrypt that traffic. It is independent of the first issue.
>
> This issue affects ScreenOS 6.2.0r15 through 6.2.0r18 and 6.3.0r12 through 6.3.0r20. No other Juniper products or versions of ScreenOS are affected by this issue.
>
> There is no way to detect that this vulnerability was exploited.

While both Juniper's advisory and the CVE itself are short on details, comparison of the binaries for the vulnerable and patched versions reveal that the relevant change to the code is a change in the value of Q and the corresponding test vectors [3]. The natural inference, therefore, is that the attacker changed Q away from Juniper's original version (which is itself different from the default Q in the standard) and that the patched version changes it back. What makes this even more interesting is that — as noted above — even a Q value for which the attacker knows the discrete log should not lead to a passive decryption vulnerability because the output is supposed to be filtered through the ANSI X9.31 PRNG. This obviously raises serious questions about the accuracy of Juniper's 2013 description of their system, specifically:

1. Why does a change in Q result in a passive VPN decryption vulnerability?
2. Why doesn't Juniper's use of X9.31 protect their system against compromise of Q?
3. What is the history of the PRNG code in ScreenOS?
4. How was Juniper's Q value generated?
5. Is the version of ScreenOS with Juniper's authorized Q vulnerable to attack?

We explore the answers to these questions in the following sections.

Chapter 1 is a partial reprint of material submitted to Usenix Security 2016 for publication. Research was a collaboration with Stephen Checkoway, Shaanan Cohney, Christina Garman, Matthew Green, Nadia Heninger, Eric Rescorla, Hovav Shacham, and Ralf-Philipp Weinmann. The thesis author was a primary researcher of this work.

Chapter 2

ScreenOS

Listing 2.1: The core ScreenOS 6.2 PRNG subroutines.

```
 1 void prng_reseed(void) {
 2   blocks_generated_since_reseed = 0;
 3   if (dualec_generate(prng_temporary, 32) != 32)
 4     error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
 5   memcpy(prng_seed, prng_temporary, 8);
 6   prng_output_index = 8;
 7   memcpy(prng_key, &prng_temporary[prng_output_index], 24);
 8   prng_output_index = 32;
 9 }
10 void prng_generate(void) {
11   int time[2];
12 
13   time[0] = 0;
14   time[1] = get_cycles();
15   prng_output_index = 0;
16   ++blocks_generated_since_reseed;
17   if (!one_stage_rng())
18     prng_reseed();
19   for (; prng_output_index <= 0x1F; prng_output_index += 8) {
20     // FIPS checks removed for clarity
21     x9_31_generate_block(time, prng_seed, prng_key, prng_block);
22     // FIPS checks removed for clarity
23     memcpy(&prng_temporary[prng_output_index], prng_block, 8);
24   }
25 }
```

2.1 The ScreenOS Random Number Generator

In this section, we describe the results of our analysis of the ScreenOS 6.2 PRNG cascade subroutines.[1]

Listing 2.1 shows the decompiled source code for the ScreenOS PRNG version 6.2.0r1. Note that identifiers such function and variable names are not present in the binary; we assigned these names based on analysis of the apparent function of each symbol. Similarly, specific control flow constructs are not preserved by the compilation/decompilation process. For instance, the `for` loop on line 19 may in fact

[1]ScreenOS 6.3 is identical.

be a `while` loop or some other construct in the actual Juniper source. Decompilation does, however, preserve the functionality of the original code. For clarity, we have omitted FIPS checks that ensure that the ANSI X9.31 generator [1, Appendix A.2.4] has not generated duplicate output.

Superficially, the ScreenOS implementation appears consistent with Juniper's description: When `prng_generate()` is called, it first potentially reseeds the X9.31 PRNG state (lines $16 - 18$) via `prng_reseed()`. When `prng_reseed()` is called, it invokes the Dual EC DBRG to fill the 32-byte buffer `prng_temporary`. From this buffer, it extracts a seed and cipher key for the ANSI X9.31 generator. Once the X9.31 PRNG state is seeded, the implementation then generates 8 bytes of X9.31 PRNG output at a time (line 21) into `prng_temporary`, looping until it has generated 32 bytes of output (lines 19–24), using the global variable `prng_seed` to store the ANSI X9.31 seed state, updating it with every invocation of `prng_generate_block()`.

However, upon closer inspection, the behavior of the generator is subtly different. This is due to two coupled issues: First, `prng_reseed()` and `prng_generate_blocks()` share the static buffer `prng_temporary`; secondly, when `prng_reseed()` is invoked, it fills `prng_temporary` (line 3) and then sets the static variable `prng_output_index` to 32 (the size of the Dual EC output).[2] Unfortunately, `prng_output_index` is *also* the control variable for the loop that invokes the ANSI X9.31 PRNG in `prng_generate()` at line 19. The consequence is that whenever the PRNG is reseeded, `prng_output_index` is 32 at the start of the loop and therefore no calls to the ANSI X9.31 PRNG are executed. Thus, Dual EC output is emitted directly from the `prng_generate()` function.

[2]The global variable reuse was first publicly noted by Willem Pinckaers on Twitter `https://twitter.com/_dvorak_/status/679109591708205056`, retrieved February 18, 2016.

Another oddity is that in the default configuration, `one_stage_rng()` always returns true so X9.31 is reseeded on *every* call. There is an undocumented ScreenOS command, `set key one-stage-rng`, which is described by a string in the command-parsing data-structure as “Reduce PRNG to single stage.” Invoking this command effectively disables reseeding until this setting is changed.

Ironically, when combined with the cascade bug described above, disabling reseeding introduces a different security vulnerability: because the first block emitted after reseed is the same as the data used for future blocks of the ANSI X9.31 PRNG, an attacker who is lucky enough to observe an immediate post-seed output can predict the rest of the PRNG stream until the next reseed *even without knowing* $\log_P Q$.[3]

Interestingly, had `prng_output_index` not been used in `prng_reseed`, the reuse of `prng_temporary` would be safe. As described in section 3.1, the index variable used in the for loop in `prng_generate` changed from a local variable to the `prng_output_index` global variable between the final version of ScreenOS 6.1.0 and the first version of 6.2.0.

[3]There are technical obstacles to overcome. X9.31 uses the current time (parameter DT in the specification; implemented as the processor cycle counter in ScreenOS) as an input to the PRNG. As long as the time value can be guessed (or brute forced), the X9.31 generator’s output can be predicted. As `one-stage-rng` is off by default and this command that enables is is undocumented, we did not study this vulnerability in depth.

2.2 Interaction with IKE

As suggested by the exploit description, the primary concern with a Dual EC implementation is that an attacker may be able to use public information emitted by the PRNG to extract the Dual EC internal state, and use this to predict future secret values. Because ScreenOS is not only a firewall but also a VPN device, the natural target is IKE (Internet Key Exchange) [15, 20], the key establishment protocol used for IPsec [24]. Surprisingly, the existence of a Dual EC generator does not by itself imply that Juniper's IKE implementation is itself exploitable, even in conditions where the attacker knows the Dual EC discrete log. There are a number of parameters that affect both the feasibility and cost of such an attack.

2.2.1 Overview of IKE

IKE (and its successor IKEv2) is a traditional DH-based handshake protocol in which two endpoints (dubbed the *initiator* and the *responder*) establish a *Security Association* (SA) consisting of parameters and a set of traffic keys which can be used for encrypting traffic. Somewhat unusually, IKE consists of two phases:

- *Phase 1 (IKEv1)/Initial Exchange (IKEv2)*: Used to establish an "IKE SA" which is tied to the endpoints but not to any particular class of non-IKE network traffic.

- *Phase 2 (IKEv1)/CREATE_CHILD_SA (IKEv2)*: Used to establish SAs which protect non-IKE traffic (typically IPsec). The IKE messages for this phase are protected with keys established in the first phase. This phase may be run multiple times with the same phase 1 SA in order to establish multiple SAs (e.g., for different IP host/port pairs), but as a practical matter many VPN connections compute only one child SA and use it for all traffic.

For simplicity, we will use the IKEv1 terminology of phase 1/phase 2 in the rest of this document.

IKE messages are composed of a series of "payloads" such as KE (key exchange), Ni (initiator nonce), Nr (responder nonce), etc.

The first IKE phase consists of a Diffie–Hellman exchange in which both sides exchange DH shares and a nonce, which are combined to form the derived keys. The endpoints may be authenticated in a variety of ways including a signing key and a statically configured shared secret. The second IKE phase may involve a DH exchange but may also just consist of an exchange of nonces, in which case the child SA keys are derived from the shared secret established in the first phase.

At this point, we have a conceptual overview of how to attack IKE: Using the nonce in the first phase, reverse Dual EC to compute the PRNG state; predict the DH private key and use that to compute the DH shared secret for the IKE SA; using the keys derived from the IKE SA, decrypt the second phase traffic to recover the peer's nonce and public key (in the best case, the local nonce and public key can be predicted); use those to compute the shared secret for the second phase SA and thereby the traffic keys. Use those keys to decrypt the VPN traffic.

However, while this is straightforward in principle, there are a number of practical complexities and potential implementation decisions which could make this attack easier or more difficult (or even impractical) as described below.

2.2.2 Nonce Size

The first question we must examine is whether the attacker ever gets a complete Dual EC block. As Checkoway et al. [7] describe in detail, it is only practical to exploit Dual EC if provided with nearly a complete point output. As specified, Dual EC emits only 30 bytes of the 32-byte point, which requires the attacker to try approximately half of the remaining 2^{16} values to find the state, and the work factor goes up exponentially with the number of missing bytes, so exploitation rapidly becomes impractical the less of the point the attacker has.

Many reasonable implementation strategies could result in an attacker obtaining only small fractions of a point. For example, unlike TLS, IKE has a variable-length nonce, which is required to be between 8 and 256 bytes in length [15, Section 5]. If a nonce length below 30 bytes were used, it could significantly increase the amount of work required to recover the Dual EC state

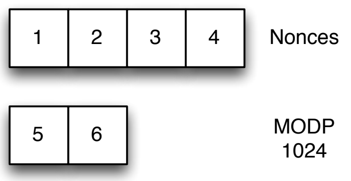
However, as of version 6.2 ScreenOS uses a 32-byte nonce made from two successive invocations of Dual EC, with the first supplying 30 bytes and the second supplying 2 bytes. As described above, this is nearly ideal from the perspective of the attacker because it can use the first 30 bytes (the majority of the point) to determine possible states, and then narrow the results by checking which states produce the correct value for the remaining two bytes. In practice, this usually results in 1 to 3 possible states.

2.2.3 Nonces and DH Keys

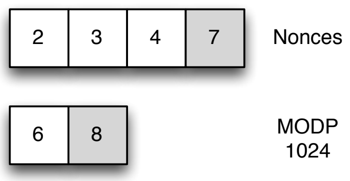
Although the IKE messages contain both a nonce and a DH share our analysis of Juniper's IKE implementation indicates that the KE payload containing the DH share is encoded *before* the NONCE payload. If (as is natural), the keys and nonces are generated in the same order as they are encoded, then it will not be possible to use the NONCE from one connection to attack that same connection. This is because Dual EC state recovery only allows you to predict future values, not recover past values. While not necessarily fatal to the attacker, because nonces generated in one connection might be used to predict the DH private keys generated in some subsequent connection; this would not be ideal from the attacker's perspective, especially if connection establishment is infrequent.

Conveniently for the attacker, however, ScreenOS also contains a pre-generation feature that maintains a pool of nonces and DH keys which can then be used in new IKE connections rather than generating them in the critical path (i.e., during the handshake). The pooling mechanism is quite intricate and appears to be designed to ensure that enough keys are always available while avoiding consuming too much run time on the device.

Summarized briefly, independent FIFO queues are maintained for nonces, each finite field DH group (MODP 768, MODP 1024, MODP 1536, and MODP 2048), and (in version 6.3) each elliptic curve group (ECP 256 and ECP 384). The sizes of these queues depend on the number of VPN configurations which have been enabled for any given group. For instance, if a single configuration is enabled for a group then that group will have queue size of 2 and disabled groups have a queue size of 0. The size of the nonce queue is set to be twice the aggregate size of all of the DH queues. So, for instance, if only the MODP 1024 group is configured, then the initial queue size will be (MODP 1024=2, nonce=4). Or, if two VPN configurations are set to use MODP 1024 and one configuration is set to use MODP 1536, initial queue size will be (MODP



(a) At system startup



(b) After a DH exchange

Figure 2.1: Nonce queue behavior during an IKE handshake.

Numbers indicate generation order, and values generated after the handshake are highlighted. During a DH exchange, outputs 1 and 5 are used as the nonce and key, and new outputs are generated to fill the end of the queue.

1024=4, MODP 1536=2, nonce=12). At initial startup time, the system completely fills all the queues to capacity and then sets a timer that fires every second to refill the queues if any values have been used.[4] If a nonce or a DH key is ever requested when the queue is empty, then a fresh value is generated on the fly.

Importantly, the queues are filled in priority order with nonces being the highest priority followed by the groups in descending order of cryptographic strength (ECP 384 down to MODP 768). This means that in many (but not all) cases, the nonce for a given connection will precede the keys for that connection in the random number sequence.

Figure 2.1 shows a (somewhat idealized) sequence of generated values[5], with the numbers indicating the order in which they were generated before and after an IKE DH exchange. Figure 2.1a shows the situation after startup: The first four values

[4]Note: only one value is generated per second, so if several values are used, it takes some time to refill the queue.

[5]For simplicity, we represent multiple consecutive invocations of the RNG as a single value and ignore invocations of the RNG for non-IKE purposes. In addition, because the queues are refilled asynchronously with respect to the IKE exchanges, there is a race condition between values being consumed and being refreshed. The pattern shown here and below is the result of assuming that the timer fires between handshakes. If it fires more frequently (i.e., between each DH and nonce encoding), then the nonces become even and the DH shares become odd. Mixed patterns are also possible.

are used to fill the nonce queue and the next two values are used to generate the DH shares. Thus, when the exchange happens, it uses value 1 for the nonce and value 5 for the key, allowing the attacker to derive the Dual EC state from value 1 and then compute forward to find the DH private key. After a single DH exchange, which requires DH key and one nonce, the state is as shown in Figure 2.1b, with the new values shaded. Note that the next-in-line values continue to have the property that the nonce was generated before the DH share. Because nonce computation is prioritized over key generation, in this simple configuration where you have a single DH group that is used for every handshake, then as long as handshakes are done reasonably slowly (giving the background task enough time to fill the queue) the nonce used for a given handshake will always have been generated prior to the DH key for that handshake. Of course, if a large number exchanges are run in succession (i.e., outpacing the background task) it is possible to exhaust both queues entirely, at which point the request for a key or nonce will cause the value to be generated immediately, resulting in the DH being computed before the nonce.

2.2.4 Non-DH Phase 2 Exchanges

As noted above, the phase 2 exchange need not include a new DH exchange; implementations can simply do a nonce exchange and generate fresh keys (although Juniper's documentation recommends doing DH for phase 2 as well) [18, Page 72]. In this case, IKE will consume an additional nonce from the nonce queue but not a new DH key from the DH key queue. In the case where endpoints do a single phase 1 exchange and then a phase 2 exchange, with only the former doing DH, then setting up a VPN connection setup consumes two nonces and one DH key. However, because the nonce queue is twice as large as the DH queue, as long as the refill timer fires reasonably often with respect to the handshakes it is not possible to exhaust the nonce queue (thus causing a fresh PRNG value to be generated) while there is still a stale DH value in the DH queue. Note that if the nonce and DH queues were the same size, then non-DH phase 2 exchanges would frequently cause keys to be stale with respect to the nonce.

In addition, if *multiple* non-DH phase 2 exchanges are done within a single phase 1 exchange, then it is possible to empty the nonce queue while there are still values in the DH queue. In this case, it will only be possible to decrypt connections established using those values if the attacker has recorded previous nonces, rather than decrypting a connection in isolation. Similarly, the current nonces could be used to decrypt future connections but not the connections they are transmitted with.

2.2.5 Multiple Groups

In addition, if the device is configured to use multiple groups, then it is possible to have the shares for one group become stale with respect to the nonces queue, as shown in Figure 2.2, which shows the result of eight MODP 2048 exchanges on the queues. The shaded MODP 1024 values were all generated before any of the remaining nonces. If the attacker starts listening at this point and observes a MODP 1024 exchange, he will not be able to decrypt it.

2.2.6 Recovering traffic keys

As described above, IKE comes in two versions (IKEv1 and IKEv2) which are slightly different. Furthermore, each version uses a somewhat unusual two-phase

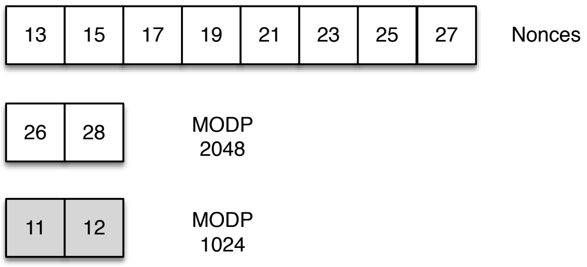


Figure 2.2: Queue state after 8 MODP 2048 exchanges.

Numbers indicate generation order, and stale values are highlighted. If several connections have been made to the same DH group, the other DH group can grow stale as all nonces that were generated before those keys are used up.

approach to protecting traffic. In this section, we describe the phases and the authentication modes that determine whether or not protected traffic can be passively decrypted.

IKEv1, phase 1. IKEv1 defines four authentication modes for phase 1: digital signatures, two modes using public-key encryption, and preshared keys [15, Section 5]. Although the details vary, each mode computes a shared secret, $SKEYID$ derived from secret values (e.g., nonces and Diffie–Hellman keys) exchanged in the handshake. Next, the encryption keying material, $SKEYID_e$ is derived from $SKEYID$ and finally the traffic keys used to protect phase 2 are derived from that in an algorithm-specific manner. Authentication keying material is derived in a similar fashion, but since an adversary is primarily concerned with decryption rather than authentication, we omit discussion of authentication below except as it relates to decryption.

- **Authentication with digital signatures.** In this mode, the initiator and responder nonces and DH public keys are exchanged in the clear. Starting with the responder's nonce, an attacker who can recover the responder's DH private key has all of the material necessary to compute *SKEYID* and thus traffic keys.

- **Authentication with public key encryption.** IKEv1 defines two public-key encryption modes for authentication. The revised mode uses half the number of public-key encryptions and decryptions the other mode uses, but are otherwise similar. In these modes, the DH public keys are exchanged in the clear but each peer encrypts its nonce using the other's public key. These modes require the initiator to know the responder's public key prior to the handshake. Each peer decrypts the other's nonce and computes *SKEYID*. Since nonces are encrypted, even if an attacker can recover the responder's nonce (e.g., by capturing a nonce in the clear from a previous connection, recovering the Dual EC state, and walking the generator forward), the initiator's nonce is also encrypted, thus stopping the attack.

- **Authentication with preshared keys.** In this mode, a preshared key needs to be established out of band. The DH public keys and nonces are exchanged in the clear. The encryption keying material, $SKEYID_e$ is derived from the preshared key, the nonces, and the DH keys. An attacker who can recover a DH private key can perform an offline attack on the preshared key. Depending on the strength of the PSK, this process of recovering it may be trivial or may be computationally intractable.[6]

IKEv1, phase 2. After phase 1 completes, there is a second phase, called Quick Mode, which involves another exchange of nonces and, optionally, another DH exchange

[6]Anecdotally, the preshared keys used in practice are often quite weak. For example, FlyVPN's "How To Setup L2TP VPN On Android 4" instructs the user to "Input 'vpnserver' letters into 'IPSec pre-shared key.'" `https://www.flyvpn.com/How-To-Setup-L2TP-VPN-On-Android-4.html`, retrieved February 18, 2016.

for forward secrecy [20]. As the messages for phase 2 are protected by the keys established during phase 1, there is no additional encryption. Thus, an attacker who has successfully recovered the phase 1 keys can decrypt phase 2 messages. At this point, if another DH key exchange is used, the attacker can either run the Dual EC-state-recovery attack again or simply walk the Dual EC generator forward to recover the DH private key. If only nonces are exchanged, then no additional work is necessary. In either case, the attacker can compute the traffic keys and recover plain text.

IKEv2, phase 1. A connection in IKEv2 begins by exchanging two request/response pairs which form the initial exchange. The first pair of messages, called IKE_SA_INIT, exchange DH public keys and nonces in the clear. The peers use these to compute a shared secret, *SKEYSEED*, from which all traffic keys are derived. These keys are used to protect the following messages.

This first exchange contains all of the information necessary for the attacker to recover the Dual EC state and compute a DH private key and thus derive *SKEYSEED*. This stands in contrast to IKEv1 where the authentication mode influences key derivation and hence, exploitability.

The second exchange, called IKE_AUTH, is encrypted using keys derived from *SKEYSEED* and is used to authenticate each peer, but plays no role in decryption. At this point, a child security association (CHILD_SA) is set up which can be used for protecting VPN traffic.

IKEv2, phase 2. IKEv2 does contain a second phase, called CREATE_CHILD_SA, which can be used to create additional child security associations. One use of this phase is periodic rekeying. The use of a second phase is optional.[7]

Similar to IKEv1's second phase, nonces and, optionally, DH public keys are exchanged. As before, when DH keys are used, an attacker may either perform the Dual EC-state-recovery attack a second time or walk the generator forward.

[7] "The second request/response (IKE_AUTH) transmits identities, proves knowledge of the secrets corresponding to the two identities, and sets up an SA for the first (and often only) AH and/or ESP CHILD_SA" [20].

2.3 Exploiting the Vulnerability against IKE

To validate the attacks we describe above, we purchased a Juniper NetScreen SSG-500M VPN device, and modified the firmware version 6.3.0r12 in a manner similar to the 2012 attack. This required us to generate a point Q for which we know the trapdoor $(\log_P Q)^{-1}$, and to modify the Dual EC Known Answer Test (KAT) correspondingly. To install the firmware on the device, we further modified a non-cryptographic checksum contained within the header of the firmware.[8]

Using the new firmware, we next configured the device with three separate VPN gateways: (1) configured for IKEv1 with a PSK, (2) configured for IKEv1 with a 1024-bit RSA signing certificate, and (3) configured for IKEv2 with a PSK. For each configuration, we initiated VPN connections to the box using strongSwan [34]. By capturing the resulting traffic, we were then able to extract the nonces in the IKE handshakes and run the Dual EC attack to recover the state of the random number generator for each connection. As previously discussed, since the 32-byte nonces consist of the concatenation of two consecutive 30-byte Dual EC blocks, truncated to 32-bytes, we used the first 30 bytes of the nonce to recover a potential state value, and then confirmed this guess against the remaining 2 bytes of the nonce.

From this point, we generated a series of Dual EC outputs to obtain a private exponent consistent with the Diffie–Hellman public key observed in the traffic. This required a single modular exponentiation per potential exponent x, followed by a comparison to the extracted key exchange payload value. Given the correct private exponent, we then obtained the shared secret from the initiator value, thereby

[8]If a certificate is installed on the device, firmware updates require the presence of a valid digital signature on the new firmware. Since we did not have a certificate installed, we were able to omit this signature.

determining the DH shared secret g^{xy}. Given the Diffie–Hellman shared secret, we implemented the remaining elements of the IKEv1 and IKEv2 standards [15, 20] in order to calculate the Phase 1 (Aggressive Mode) keying material (for IKEv1) and the corresponding IKE_SA_INIT/IKE_AUTH keying material (for IKEv2). This information encrypts the subsequent handshake messages, and is itself used to calculate the key material for subsequent payloads, including Encapsulated Secure Payload (ESP) messages. A challenge in the IKEv1 PSK implementation is the need to incorporate an unknown PSK value into the PRF used to calculate the resulting key material. For our proof of concept implementation we used a known PSK, however without knowledge of this value, an additional brute-force or dictionary attack step would have been required. No such problem exists for the IKEv1 certificate connections, or for IKEv2 PSK.

Using the recovered key material, we next decrypt the remaining traffic, which in each case embeds a second Diffie–Hellman handshake with additional nonces and Diffie–Hellman ephemeral public keys. Since this handshake is also produced from the same generator, we can simply wind the generator forward (or restart with a nonce drawn from the second phase handshake) to recover the corresponding Diffie–Hellman private keys. This new shared secret can then be used to calculate the resulting key material. All subsequent traffic that we see and decrypt utilized the Encapsulating Security Payload (ESP) protocol [23] in tunnel mode.

2.4 Passively detecting Juniper ScreenOS

An adversary who knows the Dual EC Q parameter — either Juniper's non-standard point or the point that was introduced into ScreenOS unbeknownst to Juniper — may wish to detect vulnerable versions of ScreenOS by passively watching network traffic. In theory, such an adversary has several avenues open to it. The easiest approach is to attempt the attack on every VPN connection to see if the attack is successful. Alternatively, the adversary could attempt to fingerprint VPN boxes and only perform the attack on connections that match.

Dual EC is known to have a small, but nonnegligible, bias. In particular, Schoenmakers and Sidorenko [32] and Gjøsteen [12] give a procedure to distinguish 30-byte blocks generated uniformly at random from those generated by Dual EC. The basic idea is to count how many points on the curve have x-coordinates that agree with the 30-byte block in their least-significant 30-bytes. In both the uniformly at random case and the Dual EC case, the number of points on the curve that match follow a normal distribution. In order to use the this distinguisher, one needs to see a sufficient number of 30-byte blocks (in the form of IKE nonces) to state with high confidence that the blocks came from one distribution or the other.

We empirically computed the parameters of these two distributions to see how difficult this task is. We generated 2 million 30-byte blocks using Dual EC and an additional 2-million blocks uniformly at random and performed the point counting. We estimate the distributions' parameters by fitting the data using maximum likelihood estimation. The results are not encouraging. When generated uniformly at random, the number of points on the curve that agree with the generated block have parameters $\mu = 65536.02$ and $\sigma = 256.05$. When generated using Dual EC, the parameters are

$\mu = 65536.78$ and $\sigma = 256.06$. This approach is unlikely to work without seeing tens or hundreds of thousands of connections.

Interestingly, ScreenOS's Dual EC implementation has a bug that makes the adversary's job much easier. ScreenOS contains a customized version of OpenSSL and uses OpenSSL's elliptic curve and arbitrary-precision (BIGNUM) routines to implement Dual EC. The OpenSSL function to convert a BIGNUM to an array of bytes is `BN_bn2bin()`. Due to a design defect in OpenSSL's API, there is no way to correctly use `BN_bn2bin` without first determining how many bytes it will use — using `BN_num_bytes()` — and zero-padding, and only then using `BN_bn2bin()`:[9]

```
int size = BN_num_bytes(x);
memset(buffer, 0, 30 - size);
BN_bn2bin(x, buffer + size);
```

ScreenOS's Dual EC implementation omits the zero-padding when converting from BIGNUMs to binary output. The upshot is that neither the first nor the thirty-first byte of a nonce will ever be a zero byte.

Thus, if the adversary ever sees a zero byte in either position, it can conclude that the implementation is not Juniper's Dual EC.

If the nonces are generated uniformly at random, then we expect each of these bytes to be zero with (independent) probability 1/256. Thus, the probability that after n nonces without zeros in those positions, the nonce was generated uniformly at random is $1 - (255/256)^{2n}$.

This bug does not affect the exploitability of the Dual EC generator; however, it can lead to a few additional potential internal states to check as described in

[9]This defect was corrected quite recently, years after the version of OpenSSL ScreenOS uses was written. `https://mta.openssl.org/pipermail/openssl-commits/2016-February/003520.html`

Section 1.2.

Chapter 2 is a partial reprint of material submitted to Usenix Security 2016 for publication. Research was a collaboration with Stephen Checkoway, Shaanan Cohney, Christina Garman, Matthew Green, Nadia Heninger, Eric Rescorla, Hovav Shacham, and Ralf-Philipp Weinmann. The thesis author was a primary researcher of this work.

Chapter 3

Discussion

Listing 3.1: The core ScreenOS 6.1 PRNG subroutine.

```
void prng_generate(char *output) {
  unsigned int index;
  int time[2];

  index = 0;
  // FIPS checks removed for clarity
  if ( blocks_generated_since_reseed++ > 9999 )
    prng_reseed();
  // FIPS checks removed for clarity
  time[0] = 0;
  time[1] = get_cycles();
  do {
    // FIPS checks removed for clarity
    prng_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&output[index], prng_output_block, min(20-index, 8));
    index += copy_amount;
  } while ( index <= 19 );
}
```

3.1 Discussion

A significant amount of attention has been paid to the 2012 compromise of Juniper's ScreenOS source code by unknown parties. In this paper we have shown that the vulnerabilities announced by Juniper can be traced largely to the pre-existing design of Juniper's ScreenOS random number generator. Specifically, we argue that Juniper's design is exploitable due to a series of deliberate design decisions, accidents, and oversights on the part of the ScreenOS developers.

Below we review each of the conditions that are required to produce an exploitable PRNG in the Juniper system:

1. **Implementation of Dual EC.** The cascade design of Juniper's double PRNG, which employs Dual EC to seed the RNG on each call seems a surprising choice, given the performance limitations of Dual EC. Notably, the transition from

ScreenOS 6.1 (ANSI only) to 6.2 (Dual EC and ANSI) involved the addition of a *nonce pre-generation* queue to the existing DH key queues.[1] One potential motivation for this change could be the additional security assurance provided by Dual EC. However, it is surprising to note that Juniper did not seek FIPS certification of the Dual EC generator, despite the fact that following the deprecation of the ANSI X9.31 generator on January 1, 2016, it would have been the only FIPS-certified PRNG in their product.[2]

2. **Presence of a Dual EC/ANSI cascade flaw.** Even with Dual EC present in the ScreenOS devices, the use of a cascade between Dual EC and ScreenOS should have prevented the known state recovery attacks. As detailed in Section 2.1, this protection is not available due to flaws in the cascade implementation, which allows for the exfiltration of unprocessed Dual EC output. This flaw is particularly perplexing. Compare the `prng_generate()` function in version 6.1 (Listing 3.1) with the analogous function in version 6.2 (Listing 2.1). Apart from minor changes arising from moving from generating 20 bytes at a time to generating 32 bytes at a time and always reseeding rather than reseeding based on a counter, the functions look quite similar. However, for some reason, the loop index variable was changed from a local variable to a global variable.

3. **Always reseeding.** In version 6.1, ScreenOS reseeded the X9.31 PRNG from system entropy every 10000 calls (hardcoded; see Listing 3.1). However, in version 6.2, the reseeding mechanism was repurposed to produce the cascade by

[1]To give a rough estimate of the performance difference, we implemented Dual EC and ANSI X9.31 using the same procedure used in ScreenOS and measured how long it takes to generate 32-byte blocks. Dual EC takes roughly 125 times as long as X9.31.

[2]A review of the CMVP certification lists [29] shows that all ScreenOS FIPS certification certificates have indeed been de-listed as of February 2016.

always reseeding. When combined with the cascade flaw described above, all PRNG output comes from Dual EC, increasing the probability that a specific value observed by the attacker can be used to recover PRNG state.

4. **Use of 32-byte IKE nonces.** The IKE standards do not provide a specific recommendation for nonce length, stating only that nonces should be between 8 and 256 bytes, and that nonces should be at least half the key size of the PRF used. The last version of ScreenOS without Dual EC was 6.1.0r7 and specified 20 byte nonces. In the subsequent release, ScreenOS 6.2.0r1, Juniper developers added Dual EC and modified the IKE nonce size from 20 to 32 bytes. Efficiently recovering the state of the Dual EC generator requires at a minimum 26 bytes of unprocessed PRNG output, and as discussed in Section 2.2.2, having greater than 30 bytes expedites the state recovery attack.

5. **Modifying the order of nonce and key generation.** The ScreenOS IKEv1 and IKEv2 implementations both output IKE Key Exchange prior to the IKE Nonce packet. However, this output order does not reflect the *generation* order of the same values in all versions of ScreenOS. In particular, the addition of *nonce queues* in ScreenOS 6.2.0r1 effectively guarantees that in most cases a non-loaded system will generate a nonce immediately prior to the Diffie–Hellman private key that will be used in a given handshake. In practice, this facilitates state recovery attacks that can recover secret keys (and thus enable decryption) within a single IKE handshake, significantly improving the effectiveness of passive attacks.

All told, in the course of a single version revision, Juniper made a series of changes that combined to produce a system which only required the attacker to know

Table 3.1: ScreenOS features by version.

Between versions 6.1.0 and 6.2.0, a cluster of changes were made to the PRNG and IKE subsystems. In the PRNG subsystem, the switch to (1) Dual EC + X9.31; (2) reseeding on every call; and (3) the bug in reseed that causes X9.31 to be skipped produce the necessary conditions to attack IKE. In the IKE subsystem, changing the nonce size from 20 bytes to 32 bytes moves the attack from completely impractical to nearly best-case scenario, from an attacker's point of view. The introduction of a nonce queue changes the nature of the attack such that, in the usual case, an attacker can decrypt a session based solely on that session's traffic.

Version 6.3.0 is nearly identical to 6.2.0 but supports elliptic curve Diffie–Hellman groups. In contrast to the changes between 6.1.0 and 6.2.0, this may actually make an attacker's job harder; see Section 2.2.5.

Version	PRNG	Reseed period (calls)	Reseed bug	DH queue	Nonce queue	Nonce size (bytes)	DH groups supported
6.1.0	X9.31	10000		✓		20	MODP 768, 1024, 1536, 2048
6.2.0	Dual EC + X9.31	1	✓	✓	✓	32	MODP 768, 1024, 1536, 2048
6.3.0	Dual EC + X9.31	1	✓	✓	✓	32	MODP 768, 1024, 1536, 2048, ECP 256, 384

the discrete log of Q to be exploitable. See Table 3.1 for a summary of changes. For a randomly-selected Q, or a point chosen using the nothing-up-my-sleeve process proposed in ANS X9.82 [2], calculating d is likely to be infeasible. We have no way to evaluate the likelihood that some party knows d for Juniper's non-standard Q point, except to note that Juniper does not appear to have used any of the recommendations presented in the NIST standard [28]. Based on the conclusions of Juniper's 2012 vulnerability report [17], however, it does seem reasonable to assume that the 2012 attacker-generated Q' was maliciously generated.

Regardless of the causes, these ScreenOS vulnerabilities provide an important lesson for the design of cryptographic systems. By far the most attractive feature of the ScreenOS PRNG attack, from the perspective of an attacker, is the ability to significantly undermine the security of ScreenOS *without* producing any externally-detectable indication that would reveal that ScreenOS devices were vulnerable. This is in marked contrast to previous well-known PRNG failures, such as the Debian PRNG flaw, that. Indeed, the versions of ScreenOS containing an attacker-supplied parameter appear to have produced output that was cryptographically indistinguishable from the output of previous versions. Thus, preventing any testing or measurement from discovering the issue.

In Section 1.3, we asked four questions which we reproduce here, along with their answers:

1. Why does a change in Q result in a passive VPN decryption vulnerability? *If an attacker is able to replace Q with a value of their choice, they can predict the (EC)DH keys which are used to establish the VPN traffic keys, thus decrypting VPN traffic.*

2. Why doesn't Juniper's use of X9.31 protect their system against compromise of Q? *In the default configuration, the X9.31 PRNG is never run, and Dual EC output is sent directly to the network in the form of nonces. These nonces can be used to extract the Dual EC state and predict future outputs.*

3. What is the history of the PRNG code in ScreenOS? *Version 6.1 contained a conventional X9.31-based PRNG which was replaced with a two-stage PRNG based on Dual EC. That same version made a number of other changes to the PRNG and IKE code which combine to make ScreenOS vulnerable to a Q-replacement attack.*

4. How was Juniper's Q value generated? *We are unable to answer this question. In normal cases, the distribution of a maliciously-generated Q is identical to the distribution of a random Q. There do, however, exist techniques for* verifiably

generating safe Q points such that no attacker is likely to know the discrete log (one such procedure is described in [2]). If Juniper employed such a technique, it is still possible that a thorough review of historical documentation might provide evidence that such a technique was used.

5. Is the version of ScreenOS with Juniper's authorized Q vulnerable to attack? *We are unable to answer this question definitively. As demonstrated in this paper, an attacker who knows the discrete log of Juniper's Q would be able to decrypt traffic. However, because we do not know the details of Juniper's Q-generation procedure, we are unable to determine if such attackers exist.*

3.2 Related Work

Dual EC. The history of the Dual EC random number generator was described by Checkoway et al. [7]. By 2006, it was already clear that the generator output has biases in its output that make it unsuitable for deployment, through work by Gjøsteen [12] and by Schoenmakers and Sidorenko [32]. Shumow and Ferguson's presentation at the Crypto 2007 rump session [33] further made clear that someone in possession of the discrete logarithm of Q to base P and who saw raw output from the generator would be able to reconstruct its internal state and predict all future outputs. Nevertheless, Dual EC was adopted as part of NIST's SP 800-90A standard [4], and was not withdrawn until 2015 [5], following reporting based on the Snowden documents [30] that suggested that the Dual EC backdoor might be intentional. A remarkable presentation by John Kelsey gives a postmortem of Dual EC standardization from NIST's perspective [21].

Our analysis in Section 2.1 shows that Juniper adopted Dual EC in 2008. In 2013, NIST's reopening SP 800-90A for comments led Juniper to publish a knowledge base article explaining that ScreenOS uses Dual EC, but "in a way that should not be vulnerable to the possible issue that has been brought to light," because of the custom Q and because Dual EC output is filtered through X9.31 [19]. As our analysis shows, at the time that Juniper made this statement, the Q generator shipping in ScreenOS was the one introduced in the unauthorized 2012 change. In January 2016, Juniper announced that it would remove Dual EC from its ScreenOS products in "the first half of 2016" [35].

Randomness failures. Many instances of randomness failures in widely deployed systems have been reported. In 1996, Goldberg and Wagner showed that the Netscape browser seeded its PRNG insecurely, allowing SSL traffic to be decrypted [13].

Between 2006 and and 2008, Linux systems running the Debian distribution or its derivatives (including Ubuntu) shipped a modified version of the OpenSSL library that failed to incorporate entropy from the kernel into its own entropy pool. The available entropy was then low enough under normal conditions that the keys that affected systems generate could be exhaustively enumerated and identified over the network [36].

Heninger et al. [16] performed a pairwise GCD on RSA moduli obtained from scanning the IPv4 address space, finding many shared factors and weak keys; the root cause of the failure was the lack of entropy available shortly after boot in many network devices. Kim et al. [25] showed that a related problem affected OpenSSL on Android.

Bernstein et al. showed that randomness failures in smart cards allowed private keys to be recovered using lattice attacks [6].

Design of PRNGs. A line of work beginning with Kelsey et al. [22] and continuing to today [8, 9]. has sought to formalize the security desiderata for PRNGs used as part of cryptographic systems, and to evaluate deployed PRNGs against these desiderata. Gutterman et al. [14] and, later, Lacharme et al. [26] analyzed the Linux randomness system, and Dorrendorf et al. [10] analyzed that of Windows.

As new use cases arose, the security desiderata have been revised and expanded. For example, Ristenpart and Yilek analyzed application-level randomness reuse in virtual machines whose state is reset and rolled back [31], and Everspaugh et al. [11]

extended the analysis to kernel-level randomness.

3.3 Summary

Following Juniper's disclosure of unauthorized code in their ScreenOS VPN, we reverse engineered multiple versions of ScreenOS to determine exactly what had happened. We find that while the proximal cause of the vulnerability was the replacement of the Q parameter from the Dual EC PRNG, the attack was only possible due to the interaction of a cluster of changes made by Juniper in the 6.2 version of ScreenOS released in 2008. Those changes included replacing their conventional X9.31 PRNG with a two-stage PRNG which is described as using Dual EC to seed the X9.31 PRNG; in fact, however, in the default configuration the X9.31 PRNG never executes and Dual EC values are output directly from the PRNG subsystem. Taken together with a number of changes to the IKE implementation, this PRNG structure enables an attacker who knows the discrete log of Q to passively decrypt IKE handshakes and the IPsec traffic protected with keys derived from those handshakes. We have validated the results of our binary analysis by testing a modified ScreenOS binary with our own value Q (for which we have the discrete log) and verifying that we can decrypt the results of IKEv1 and IKEv2 handshakes.

Chapter 3 is a partial reprint of material submitted to Usenix Security 2016 for publication. Research was a collaboration with Stephen Checkoway, Shaanan Cohney, Christina Garman, Matthew Green, Nadia Heninger, Eric Rescorla, Hovav Shacham, and Ralf-Philipp Weinmann. The thesis author was a primary researcher of this work.

Appendix A

Dual EC Values

Our experiments used the P-256 curve parameters. This curve is defined over F_p with $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. The curve is given in short Weierstrass form $E : y^2 = x^3 - 3x + b$, where

$$b = \texttt{5ac635d8 aa3a93e7 b3ebbd55 769886bc} \setminus$$
$$\texttt{651d06b0 cc53b0f6 3bce3c3e 27d2604b}.$$

The base point P has order n, where

$$\begin{aligned} P_x &= \texttt{6b17d1f2 e12c4247 f8bce6e5 63a440f2} \setminus \\ &\quad \texttt{77037d81 2deb33a0 f4a13945 d898c296} \\ P_y &= \texttt{4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16} \setminus \\ &\quad \texttt{2bce3357 6b315ece cbb64068 37bf51f5} \\ n &= \texttt{ffffffff 00000000 ffffffff ffffffff} \setminus \\ &\quad \texttt{bce6faad a7179e84 f3b9cac2 fc632551}. \end{aligned}$$

The official second point Q as given in the Dual EC description is

$$\begin{aligned} Q_x &= \texttt{c97445f4 5cdef9f0 d3e05e1e 585fc297} \setminus \\ &\quad \texttt{235b82b5 be8ff3ef ca67c598 52018192} \\ Q_y &= \texttt{b28ef557 ba31dfcb dd21ac46 e2a91e3c} \setminus \\ &\quad \texttt{304f44cb 87058ada 2cb81515 1e610046}. \end{aligned}$$

To implement our attacks we generated the following random constant e to compute a new point $Q' = eP$. The trapdoor is $d \equiv e^{-1} \pmod{n}$.

$$\begin{aligned}
e &= \texttt{facc5582 909e66b3 09b1a3ae 5e4d51fc} \setminus \\
& \texttt{0edbfb57 6ef8bfa9 c233b035 9f7a7b49} \\
d &= \texttt{6fc45453 894de99c 661581b0 a12087b8} \setminus \\
& \texttt{62667b78 5aaba711 6dcdcb3c b3a79afe} \\
Q'_x &= \texttt{f6c4f766 b3c61f09 e6095822 24cc8ebc} \setminus \\
& \texttt{cf4cd496 1ef780cc 02e8f09a 0efa7ca5} \\
Q'_y &= \texttt{f212c576 8d46716c 6cac4d23 ff12e8ae} \setminus \\
& \texttt{89fd9eee c83a0e83 e35db3aa de0ccb5b}.
\end{aligned}$$

Bibliography

[1] Accredited Standards Committee (ASC) X9, Financial Services. ANS X9.31-1998: Digital signatures using reversible algorithms for the financial services industry (rDSA), 1998. Withdrawn.

[2] Accredited Standards Committee (ASC) X9, Financial Services. ANS X9.82-3-2007: Random number generation, part 3: Deterministic random bit generators, 2007.

[3] Anonymized for submission. Anonymized for submission, Dec. 2015.

[4] E. Barker and J. Kelsey. NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, National Institute of Standards and Technology, 2006.

[5] E. Barker and J. Kelsey. NIST Special Publication 800-90A Revision 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, National Institute of Standards and Technology, June 2015.

[6] D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. Someren. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *ASIACRYPT '13*, pages 341–360. Springer, 2013. ISBN 978-3-642-42045-0. doi: 10.1007/978-3-642-42045-0_18.

[7] S. Checkoway, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, H. Shacham, and M. Fredrikson. On the practical exploitability of Dual EC in TLS implementations. In *Proceedings of USENIX Security 2014*, pages 319–335. USENIX Association, Aug. 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway.

[8] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, and D. Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Proceedings of CCS 2013*. ACM Press, Nov. 2013.

[9] Y. Dodis, A. Shamir, N. Stephens-Davidowitz, and D. Wichs. How to eat your entropy and have it too–optimal recovery strategies for compromised rngs. In *Advances in Cryptology–CRYPTO 2014*, pages 37–54. Springer, 2014.

[10] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the random number generator of the windows operating system. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):10, 2009.

[11] A. Everspaugh, Y. Zhai, R. Jellinek, T. Ristenpart, and M. Swift. Not-so-random numbers in virtualized linux and the whirlwind rng. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 559–574. IEEE, 2014.

[12] K. Gjøsteen. Comments on Dual-EC-DRBG/NIST SP 800-90, draft December 2005, Mar. 2006. URL http://www.math.ntnu.no/~kristiag/drafts/dual-ec-drbg-comments.pdf.

[13] I. Goldberg and D. Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal*, 1996.

[14] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[15] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), Nov. 1998. URL http://www.ietf.org/rfc/rfc2409.txt. Obsoleted by RFC 4306, updated by RFC 4109.

[16] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2362793.2362828.

[17] Juniper. 2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756), Dec. 15. URL https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713&cat=SIRT_1&actp=LIST.

[18] *Concepts & Examples ScreenOS Reference Guide: Virtual Private Networks*. Juniper Networks, rev. 02 edition, Dec. 2012. URL http://www.juniper.net/techpubs/software/screenos/screenos6.3.0/630_ce_VPN.pdf.

[19] Juniper Networks. Juniper Networks product information about Dual_EC_DRBG. Knowledge Base Article KB28205, Oct. 2013. Online: https://web.archive.org/web/20151219210530/https://kb.juniper.net/InfoCenter/index?page=content&id=KB28205&pmv=print&actp=LIST.

[20] C. Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard), Dec. 2005. URL http://www.ietf.org/rfc/rfc4306.txt. Obsoleted by RFC 5996, updated by RFC 5282.

[21] J. Kelsey. Dual EC in X9.82 and SP 800-90A. Presentation to NIST VCAT committee. Available at http://csrc.nist.gov/groups/ST/crypto-review/documents/dualec_in_X982_and_sp800-90.pdf, May 2014.

[22] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In *FSE '98*, pages 168–188. Springer, 1998. ISBN 978-3-540-69710-7. doi: 10.1007/3-540-69710-1_12.

[23] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), Nov. 2005. URL http://www.ietf.org/rfc/rfc4303.txt.

[24] S. Kent and K. Seo. Security architecture for the Internet Protocol, Dec. 2005. URL https://tools.ietf.org/html/rfc4301.

[25] S. H. Kim, D. Han, and D. H. Lee. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 659–668. ACM, 2013.

[26] P. Lacharme, A. Röck, V. Strubel, and M. Videau. The linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012. https://eprint.iacr.org/.

[27] H. Moore. CVE-2015-7755: Juniper ScreenOS Authentication Backdoor. https://community.rapid7.com/community/infosec/blog/2015/12/20/cve-2015-7755-juniper-screenos-authentication-backdoor, Dec. 2015.

[28] National Institute of Standards and Technology. NIST opens draft Special Publication 800-90A, recommendation for random number generation using deterministic random bit generators for review and comment. http://csrc.nist.gov/publications/nistbul/itlbul2013_09_supplemental.pdf, Sept. 2013.

[29] National Institute of Standards and Technology. CMVP historical validation list, Feb. 2016. URL http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-historical.htm. Retrieved February 18, 2016.

[30] N. Perlroth, J. Larson, and S. Shane. N.S.A. able to foil basic safeguards of privacy on web. *The New York Times*, September 5 2013. Online: http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html.

[31] T. Ristenpart and S. Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[32] B. Schoenmakers and A. Sidorenko. Cryptanalysis of the Dual Elliptic Curve pseudorandom generator. Cryptology ePrint Archive, Report 2006/190, 2006. URL http://eprint.iacr.org/.

[33] D. Shumow and N. Ferguson. On the possibility of a back door in the NIST SP800-90 Dual Ec Prng. Presented at the CRYPTO 2007 rump session, Aug. 2007. URL http://rump2007.cr.yp.to/15-shumow.pdf.

[34] strongSwan. strongSwan: the opensource IPsec-based VPN solution, Nov. 2015. URL https://www.strongswan.org/.

[35] B. Worrall. Advancing the security of Juniper products. Online: http://forums.juniper.net/t5/Security-Incident-Response/Advancing-the-Security-of-Juniper-Products/ba-p/286383, Jan. 2016.

[36] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In A. Feldmann and L. Mathy, editors, *Proceedings of IMC 2009*, pages 15–27. ACM Press, Nov. 2009.

[37] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006. URL http://www.ietf.org/rfc/rfc4251.txt.