The Zoom session for this class will be recorded and made available asynchronously on Canvas to registered students.
Announcements

1. Sign up to grade HW 6!

2. HW 7 is out today! Due in 1.5 weeks
Last time:
  • Authenticated key exchange and TLS

This time:
  • Random number generation
Recall: Simplified cryptographic protocol diagram

1. Use symmetric encryption to encrypt messages.
2. Use a key exchange algorithm like Diffie-Hellman to agree on a shared symmetric key.
3. Use digital signatures to authenticate other party and guarantee integrity of key exchange.
4. Use random nonces to protect against replay attacks.
Random number generation in the cryptographic protocol

1. Use symmetric encryption to encrypt messages.
2. Use a key exchange algorithm like Diffie-Hellman to agree on a shared symmetric key.
3. Use digital signatures to authenticate other party and guarantee integrity of key exchange.
4. Use random nonces to protect against replay attacks.
“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

–John von Neumann
Generating randomness for cryptography

Sources of “true” randomness:
A hardware/physical process that extracts randomness from the environment. Examples:

• Oscillating circuits
• Quantum phenomena
• Thermal noise
• Fine-grained instruction timing
Generating randomness for cryptography

Sources of “true” randomness:
A hardware/physical process that extracts randomness from the environment. Examples:

- Oscillating circuits
- Quantum phenomena
- Thermal noise
- Fine-grained instruction timing

These still have biases, and may be slow.

Environmental entropy → PRG → Crypto keys
Generating randomness for cryptography

Sources of “true” randomness:
A hardware/physical process that extracts randomness from the environment. Examples:
- Oscillating circuits
- Quantum phenomena
- Thermal noise
- Fine-grained instruction timing

These still have biases, and may be slow.

Environmental entropy \rightarrow \text{Extractor} \rightarrow \text{PRG} \rightarrow \text{Crypto keys}

A PRG requires uniformly random inputs to be secure, so we need to use something like a hash function to obtain uniform inputs.
Practical Considerations for RNGs

- **Problem:** Inputs might not be random.

  - **Solution:** Test for randomness, use extractor functions, seed cryptographic PRG.

  - **Problem:** Testing for randomness is theoretically impossible.

  - **Solution:** Do as well as you can?

  - **Problem:** Inputs might be controlled by attacker.

  - **Solution:** Seed from a variety of sources and hope attacker doesn’t control everything.

  - **Problem:** User might request output before seeding.

    - **Possible solutions:**
      1. Don’t provide output.
      2. Provide output.
      3. Raise an error flag.
Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.
Practical Considerations for RNGs

• **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

• **Problem:** Testing for randomness is theoretically impossible.
Practical Considerations for RNGs

• **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

• **Problem:** Testing for randomness is theoretically impossible.
  Solution: ... do as well as you can?
Practical Considerations for RNGs

• **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

• **Problem:** Testing for randomness is theoretically impossible.
  Solution: ... do as well as you can?

• **Problem:** Inputs might be controlled by attacker.
Practical Considerations for RNGs

• **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

• **Problem:** Testing for randomness is theoretically impossible.
  Solution: ... do as well as you can?

• **Problem:** Inputs might be controlled by attacker.
  Solution: Seed from a variety of sources and hope attacker doesn’t control everything.
Practical Considerations for RNGs

• **Problem:** Inputs might not be random.  
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

• **Problem:** Testing for randomness is theoretically impossible.  
  Solution: ... do as well as you can?

• **Problem:** Inputs might be controlled by attacker.  
  Solution: Seed from a variety of sources and hope attacker doesn’t control everything.

• **Problem:** User might request output before seeding.
Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

- **Problem:** Testing for randomness is theoretically impossible.
  Solution: ... do as well as you can?

- **Problem:** Inputs might be controlled by attacker.
  Solution: Seed from a variety of sources and hope attacker doesn’t control everything.

- **Problem:** User might request output before seeding.
  Possible solutions:
  1. Don’t provide output.
  2. Provide output.
  3. Raise an error flag.

  **Best OS design:**
  - Don’t provide output until first seeded.
  - Always provide output after.
Pros:
- Uniform
- Strenuous
- Random
- Output
RNG security properties and threat model

**Pseudorandomness:** Output cryptographically indistinguishable from random.

**Entropy Input:** RNG seeded using enough entropy so adversary can't predict seed.

1. Ensure algorithm uses available entropy.
2. Systems: Make sure algorithm is seeded in the first place.

**Prediction Resistance:** Adversary who compromises state at time $t$ can't distinguish output $t + 1$ from random.

**Backtracking Resistance:** Adversary who compromises state at time $t$ can't distinguish output $t - 1$ from random.
Real-world threats to RNGs outside of theoretical model

• State-level adversaries interfering with the design and standardization process.

• Unclear or too-permissive algorithm specifications.

• Implementers misunderstanding algorithm specification.

• Implementation bugs.
## Government standards for RNGs

<table>
<thead>
<tr>
<th>Design Type</th>
<th>FIPS 140-2</th>
<th>NIST SP800-90A</th>
<th>ISO 18031</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block cipher designs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANSI X9.31</td>
<td></td>
<td>Disallowed 2016</td>
<td></td>
</tr>
<tr>
<td>CTR DRBG</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OFB DRBG</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Hash function designs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANSI X9.62</td>
<td></td>
<td>Disallowed 2016</td>
<td></td>
</tr>
<tr>
<td>Hash DRBG</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HMAC DRBG</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Number theoretic designs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual EC DRBG</td>
<td></td>
<td></td>
<td>✓ (?)</td>
</tr>
<tr>
<td>Micali Schnorr DRBG</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext
- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext
- Repeated public keys
  - RSA keys with common factors
    - Allows secret key recovery from public keys
- Repeated (EC)DSA signature nonces
  - Allows secret key recovery from signatures and public key
Cryptographic failures with bad randomness

• Repeating IVs/keys with stream cipher encryption
  • XOR of ciphertext reveals information about plaintext

• Repeating nonces with AES-GCM
  • Allows key recovery from ciphertext
Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext

- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext

- Repeated public keys
Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext

- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext

- Repeated public keys

- RSA keys with common factors
  - Allows secret key recovery from public keys
Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext

- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext

- Repeated public keys

- RSA keys with common factors
  - Allows secret key recovery from public keys

- Repeated (EC)DSA signature nonces
  - Allows secret key recovery from signatures and public key

\[
\begin{align*}
s_1 &= k^{-1} (h_1 + dr) \mod n \\
s_2 &= k^{-1} (h_2 + dr) \mod n \\
s_1 - s_2 &= k^{-1} (h_1 - h_2) \\
1. & \text{ Solve } h - k. \\
2. & \text{ Solve for } d. \text{ (secret key)}
\end{align*}
\]
Netscape SSL RNG Vulnerability [Goldberg Wagner 1996]

**Underlying cause:** Seeding PRNG with insufficient entropy.

```c
global variable seed;

RNG_CreateContext()
(seconds, microseconds) = time of day; /* Time elapsed since 1970 */
pid = process ID; ppid = parent process ID;
a = mklcpr(microseconds);
b = mklcpr(pid + seconds + (ppid << 12));
seed = MD5(a, b);

mklcpr(x) /* not cryptographically significant; shown for completeness */
return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

RNG_GenerateRandomBytes()
    x = MD5(seed);
    seed = seed + 1;
    return x;

global variable challenge, secret_key;

create_key()
    RNG_CreateContext();
    ...
    challenge = RNG_GenerateRandomBytes();
    secret_key = RNG_GenerateRandomBytes();
```
The Debian OpenSSL Disaster

Luciano Bello, 2008

*When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability* Yilek, Rescorla, Shacham, Enright, Savage. (2009)

**Underlying cause:** Failure to seed PRNG.
OpenSSL PRNG

- Seed: /dev/urandom, pid, time()
- Update: time() (in seconds)
- Mixing function: SHA-1
- Output: SHA-1 hash of state.
/ * state[st_idx], ..., state[(st_idx + num - 1) % STATE_SIZE] */

/* state[st_idx], ..., state[(st_idx + num - 1) % STATE_SIZE]
 * are what we will use now, but other threads may use them
 * as well */

md_count[1] += (num / MD_DIGEST_LENGTH) + (num % MD_DIGEST_LENGTH > 0);

if (!do_not_lock) CRYPTO_w_unlock(CRYPTO_LOCK_RAND);

EVP_MD_CTX_init(&m);

for (i=0; i<num; i+=MD_DIGEST_LENGTH)
{
    j=(num-i);
    j=(j > MD_DIGEST_LENGTH)?MD_DIGEST_LENGTH:j;

    MD_Init(&m);
    MD_Update(&m,local_md,MD_DIGEST_LENGTH);
    k=(st_idx+j)-STATE_SIZE;
    if (k > 0)
    {
        MD_Update(&m,&(state[st_idx]),j-k);
        MD_Update(&m,&(state[0]),k);
    }
    else
        MD_Update(&m,&(state[st_idx]),j);

    MD_Update(&m,buf,j);
    MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
    MD_Final(&m,local_md);
    md_c[1]++;

    buf=(const char *)buf + j;

    for (k=0; k<j; k++)
    {
        /* Parallel threads may interfere with this,
           * but always each byte of the new state is
           * the XOR of some previous value of its
           * and local_md (intermediate values may be lost).
           */
        state[st_idx++]=^state[st_idx][k];
        if (st_idx >= STATE_SIZE)
            st_idx=0;
    }

    EVP_MD_CTX_cleanup(&m);
Hi,

When deubugging applications that make use of openssl using valgrind, it can show alot of warnings about doing a conditional jump based on an unitialised value. Those unitialised values are generated in the random number generator. It’s adding an unintialised buffer to the pool.

The code in question that has the problem are the following 2 pieces of code in crypto/rand/md_rand.c:

247:

        MD_Update(&m,buf,j);

467:

#ifndef PURIFY

        MD_Update(&m,buf,j); /* purify complains */
#endif

...

What I currently see as best option is to actually comment out those 2 lines of code. But I have no idea what effect this really has on the RNG. The only effect I see is that the pool might receive less entropy. But on the other hand, I’m not even sure how much entropy some unitialised data has.

What do you people think about removing those 2 lines of code?

Kurt
Debian OpenSSL weak keys, 2006–2008

RNG output dependent on pid and machine architecture.

[Durumeric Wustrow Halderman 2013]
ANSI X9.31 and the DUHK attack

Practical state recovery attacks against legacy RNG implementations Shaanan Cohney, Matthew D. Green, Nadia Heninger. CCS 2018.
The ANSI X9.31 RNG

- Uses block cipher (AES or 3DES) as a mixing function.
- On each iteration, mixes state $V_{i-1}$ with timestamp $T_i$.
- Produces output block $R_i$ and new state $V_i$. 

```
T_i → AES_K → ⊕ → AES_K → V_i

V_{i-1} → ⊕ → AES_K → R_i
```
ANSI X9.31 RNG History

- 1985: DES-based RNG standardized in ANSI X9.17
- 1992: Adopted as a FIPS standard
- 1994: Included on list of approved RNGs in FIPS 140-1
- 1998: Variant using 3DES standardized in ANSI X9.31
- 1998: Kelsey et al.: state recovery if key known
- 2004: ANSI X9.31 RNG included in FIPS 186-2
- 2005: AES-based variant published by NIST and included on FIPS 140-2 approved RNGs
- 2011: FIPS deprecates ANSI X9.31 design
- 2016: ANSI X9.31 RNG removed from FIPS 140-2
X9.31 state recovery from a known key

[Kelsey, Schneier, Wagner, Hall 1998]

If key $K$ used with block cipher is known, can recover state $V_{i-1}$ from output $R_i$ by brute forcing timestamp.

\[ T_i \xrightarrow{\text{AES}_K} V_i \]

Design flaws:

• Block cipher is invertible, so if key is known can run both forwards and backwards.
• Low-resolution timestamps alone do not provide much entropy.
• Fixed symmetric key used for many outputs increases attack surface.
X9.31 state recovery from a known key
[Kelsey, Schneier, Wagner, Hall 1998]

If key $K$ used with block cipher is known, can recover state $V_{i-1}$ from output $R_i$ by brute forcing timestamp.

Design flaws:

- Block cipher is invertible, so if key is known can run both forwards and backwards.
- Low-resolution timestamps alone do not provide much entropy.
- Fixed symmetric key used for many outputs increases attack surface.
NIST ANSI X9.31 RNG standardization failure

"For AES 128-bit key, let *K be a 128 bit key."

"This *K is reserved only for the generation of pseudo random numbers."

• Standard did not specify key should not be hard-coded.
• Fortigate FortiOS v4 hard-coded NIST test vector key (oops)
Passive X9.31 state recovery in the IPsec protocol

- Raw RNG outputs: IKE nonce, cookie.

1. Use nonce, cookie to recover RNG state.
2. Once state recovered, increment forward to recover Diffie-Hellman secret
3. Verify DH exponent against public value on the wire.
CTR DRBG
CTR DRBG

- Output: AES encryption of incrementing counter.
- State key, counter values refreshed periodically
- Optional additional entropy can be mixed in on update
Cryptanalysis of CTR DRBG
[Bernstein 2017], [Woodage and Shumow 2019]

Design flaws:

• Block cipher is invertible, so if attacker learns key can move both backwards and forwards.

• In some situations, key is reused to generate many outputs, increasing attack surface (e.g. against side-channel attacks).

• Standard does not require additional entropy to be added.
Cryptanalysis of CTR DRBG
[Bernstein 2017], [Woodage and Shumow 2019]

Design flaws:

• Block cipher is invertible, so if attacker learns key can move both backwards and forwards.
• In some situations, key is reused to generate many outputs, increasing attack surface (e.g. against side-channel attacks).
• Standard does not require additional entropy to be added.

Theoretical attack when large amounts of data is buffered:

1. Attacker compromises state key $K_t$ using a side-channel attack at time $t$
2. Attacker decrypts output $r_t$ to compute the state $V_t$
3. Attacker winds generator forward using update function
4. Attacker then predicts output
Hash DRBG and HMAC DRBG
[Woodage and Shumow 2019]

HMAC DRBG:
• Use HMAC as mixing function
• Lacks a proof of security.

Hash DRBG:
• Uses a hash function as a mixing function.
• Has a proof of security.
Dual EC DRBG

On the Practical Exploitability of Dual EC in TLS Implementations
Checkoway, Fredrikson, Niederhagen, Everspaugh, Green, Lange, Ristenpart, Bernstein, Maskiewicz, Shacham. Usenix Security 2014.
Dual EC DRBG

- Parameters: Pre-specified elliptic curve points \( P \) and \( Q \).
- Seed: 32-byte integer \( s \)
- State: \( x \)-coordinate \( x(sP) \).
- Update: Set \( s = x(tP) \), where \( t = s \oplus \) optional additional input.
- Output: 30 bytes of \( x(sQ) \).
Dual EC DRBG History

- Early 2000s: Created by the NSA and pushed towards standardization
- 2004: Published as part of ANSI X9.82 part 3 draft
- 2004: RSA makes Dual EC the default PRNG in BSAFE
- 2005: Standardized in NIST SP 800-90 draft
- 2007: Shumow and Ferguson demonstrate theoretical backdoor
- 2013: Snowden documents lead to renewed interest in Dual EC
- 2014: Practical attacks on TLS using Dual EC demonstrated
- 2015: NIST removes Dual EC from list of approved PRNGs
- 2015: *Juniper incident*
1. Attacker controls standard and constructs points with known relationship $P = dQ$.

2. Attacker gets 30 bytes of output. Attacker brute forces $2^{16}$ bits to get $2^{15}$ candidate points for $sQ$.

3. For each candidate $sQ$ attacker computes $dsQ = sP$.


5. Attacker can now predict future outputs.
2013: Snowden documents reveal NSA interference

- (TS//SI//REL TO USA, FVEY) Insert vulnerabilities into commercial encryption systems, IT systems, networks, and endpoint communications devices used by targets.
- (TS//SI//REL TO USA, FVEY) Collect target network data and metadata via cooperative network carriers and/or increased control over core networks.
- (TS//SI//REL TO USA, FVEY) Leverage commercial capabilities to remotely deliver or receive information to and from target endpoints.
- (TS//SI//REL TO USA, FVEY) Exploit foreign trusted computing platforms and technologies.
- (TS//SI//REL TO USA, FVEY) Influence policies, standards and specification for commercial public key technologies.
- (TS//SI//REL TO USA, FVEY) Make specific and aggressive investments to facilitate the development of a robust exploitation capability against Next-Generation Wireless (NGW) communications.

NY Times article pointed to Dual EC DRBG recommendation.
Exploiting Dual EC trapdoor in TLS 1.2

- Server random is raw Dual EC output.
- Attacker mounts state recovery attack on server random, winds forward for DH secret.

```
client hello: client random
[ list of cipher suites ]

server hello: server random, [cipher suite]

certificate = public RSA key + CA signatures

server kex: \( p, g, g^a \), \( \text{Sign}_{RSA_{key}}(p, g, g^a) \)

client kex: \( g^b \)

KDF(\( g^{ab} \), random) \rightarrow \( k_{mc}, k_{ms}, k_e \)

client finished: \( \text{Auth}_{k_{mc}}(\text{dialog}) \)

server finished: \( \text{Auth}_{k_{ms}}(\text{dialog}) \)

Enc_{k_e}(\text{request})
```
Table 1: Summary of our results for Dual EC using NIST P-256.

<table>
<thead>
<tr>
<th>Library</th>
<th>Default PRNG</th>
<th>Cache Output</th>
<th>Ext. Random</th>
<th>Bytes per Session</th>
<th>Adim Entropy</th>
<th>Attack Complexity</th>
<th>Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSAFE-C v1.1</td>
<td>✓</td>
<td>✓</td>
<td>✓†</td>
<td>31–60</td>
<td>—</td>
<td>$30 \cdot 2^{15}(C_v + C_f)$</td>
<td>0.04</td>
</tr>
<tr>
<td>BSAFE-Java v1.1</td>
<td>✓</td>
<td>✓</td>
<td>✓†</td>
<td>28</td>
<td>—</td>
<td>$2^{31}(C_v + 5C_f)$</td>
<td>63.96</td>
</tr>
<tr>
<td>SChannel I ‡</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
<td>28</td>
<td>—</td>
<td>$2^{31}(C_v + 4C_f)$</td>
<td>62.97</td>
</tr>
<tr>
<td>SChannel II ‡</td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>—</td>
<td>$2^{33}(C_v + C_f) + 2^{17}(5C_f)$</td>
<td>182.64</td>
</tr>
<tr>
<td>OpenSSL-fixed I *</td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>20</td>
<td>$2^{15}(C_v + 3C_f) + 2^{20}(2C_f)$</td>
<td>0.02</td>
</tr>
<tr>
<td>OpenSSL-fixed III **</td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>$35 + k$</td>
<td>$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$</td>
<td>83.32</td>
</tr>
</tbody>
</table>

* Assuming process ID and counter known. ** Assuming 15 bits of entropy in process ID, maximum counter of $2^k$. See Section 4.3. † With a library-compile-time flag. ‡ Versions tested: Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2.

Attack complexity comes from:

- How much of an output block is guaranteed to be in TLS nonce.
- Brute forcing additional entropy.
- Output cached in Bsafe and read out for TLS.
NIST re-opened SP800-90A for discussion and disallowed Dual EC DRBG in 2015.
Biggest open question after all that work:

Did anyone actually use Dual EC DRBG?
The Juniper Dual EC Incident

A Systematic Analysis of the Juniper Dual EC Incident Checkoway, Maskiewicz, Garman, Fried, Cohney, Green, Heninger, Weinmann, Rescorla, Shacham. CCS 2016.
Woah! Juniper discovers a backdoor to decrypt VPN traffic (and remote admin) has been inserted into their OS source

Important Announcement about ScreenOS®
IMPORTANT JUNIPER SECURITY ANNOUNCEMENT
CUSTOMER UPDATE: DECEMBER 20, 2015  Administrative Access (CVE-2015-7755) only affects ScreenOS 6.3.0r17 through forums.juniper.net
Diff of VPN code change

Reverse engineering shows changed values are x coords for Dual EC point Q
Juniper cascaded Dual EC with ANSI X9.31

- ScreenOS FIPS validated for ANSI X9.31, not Dual EC.
- Dual EC was supposed to seed ANSI X9.31 RNG.
- Juniper used non-default points for Dual EC

The following product families do utilize Dual_EC_DRBG, but do not use the pre-defined points cited by NIST:

1. ScreenOS*

* ScreenOS does make use of the Dual_EC_DRBG standard, but is designed to not use Dual_EC_DRBG 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 PRNG, which is the random number generator used in ScreenOS cryptographic operations.

\[
\text{Dual EC} \rightarrow \text{ANSI X9.31}
\]

What the cascade was supposed to look like
Juniper cascaded Dual EC with ANSI X9.31

- ScreenOS FIPS validated for ANSI X9.31, not Dual EC.
- Dual EC was supposed to seed ANSI X9.31 RNG.
- Juniper used non-default points for Dual EC

The following product families do utilize Dual_EC_DRBG, but do not use the pre-defined points cited by NIST:

1. ScreenOS*

   * ScreenOS does make use of the Dual_EC_DRBG standard, but is designed to not use Dual_EC_DRBG 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 PRNG, which is the random number generator used in ScreenOS cryptographic operations.

But:

There was a subtle bug. They reused some global variables.

End result: Output was raw Dual EC DRBG output.
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
    prng_output_index = 32;
}
ScreenOS RNG implementation

```c
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();    // conditional reseed
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(&prng_temporary[prng_output_index], 24);
    prng_output_index = 32;
}
```
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)  // generate Dual EC output
        error_handler("FIPS ERROR: PRNG failure, unable to reseed
", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);  // copy output
    prng_output_index = 32;
}
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block); // gen output
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);  
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
    prng_output_index = 32;
}
ScreenOS RNG implementation

void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;   // global variable
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())    // always true
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
    prng_output_index = 32;
}
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)  // global variable
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
    prng_output_index = 32;  // set to 32
}
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) { // never runs
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8); // reuses buffer
    }
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
    prng_output_index = 32;
}
void prng_generate(void) {
    int time[2];
    time[0] = 0;
    time[1] = get_cycles();
    prng_output_index = 0;
    ++blocks_generated_since_reseed;
    if (!one_stage_rng())
        prng_reseed();
    for (; prng_output_index <= 0x1F; prng_output_index += 8) {
        // FIPS checks removed for clarity
        x9_31_generate_block(time, prng_seed, prng_key, prng_block);
        // FIPS checks removed for clarity
        memcpy(&prng_temporary[prng_output_index], prng_block, 8);
    } // output is raw Dual EC output!
}

void prng_reseed(void) {
    blocks_generated_since_reseed = 0;
    if (dualec_generate(prng_temporary, 32) != 32)
        error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
    memcpy(prng_seed, prng_temporary, 8);
    prng_output_index = 8;
    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
    prng_output_index = 32;
}
Passive decryption in ScreenOS IPsec

- Use random nonces to carry out state recovery attack.
- After state recovered, then recover secret keys.
- Only attacker knows trapdoor, but can validate attack using synthetic $P$, $Q$
## ScreenOS Version History

### pre-2008
**ScreenOS 6.1.0r7**
- ANSI X9.31
- Seeded by interrupts
- Reseed every 10k calls
- 20-byte IKE nonces

### 2008
**ScreenOS 6.2.0r0**
- Dual EC → ANSI X9.31
- Reseed bug exposes raw Dual EC
- Reseed every call
- Nonces generated before keys
- 32-byte IKE nonces

### 2012
**6.2.0r15**
- Attacker changed constants.

- Attacker changed constant in 2012.
- But passive decryption already enabled in earlier release.
- Juniper’s "fix" was to reinstate original Q value. After our work they removed Dual EC completely.
Discussion

• “NOBUS” backdoors can be repurposed.

• In theory, Dual EC is a provably secure crypto backdoor. In practice, it failed.

• Don’t know how Juniper’s parameters were generated, or who wrote their Dual EC cascade.

• Juniper wasn’t certified for Dual EC, so it wasn’t on the radar of researchers who looked for vulnerable implementations. Who else are we missing?
## Summary of security proofs

Proofs from [Woodage and Shumow 2019]

<table>
<thead>
<tr>
<th>Block cipher designs</th>
<th>Security Proof</th>
<th>Should you use?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI X9.31</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CTR DRBG</td>
<td>✗</td>
<td>Maybe.</td>
</tr>
<tr>
<td>OFB DRBG</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash function designs</th>
<th>Security Proof</th>
<th>Should you use?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI X9.62</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Hash DRBG</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HMAC DRBG</td>
<td>✓*</td>
<td>✓*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number theoretic designs</th>
<th>Security Proof</th>
<th>Should you use?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual EC DRBG</td>
<td>✓*</td>
<td>✗</td>
</tr>
<tr>
<td>Micali Schnorr DRBG</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Discussion

- It is surprising that RNG security models are still in flux.

- It is surprising that NIST SP800-90A designs did not receive formal security analysis until a couple of years ago.

- Cryptographic standards should probably go through developer usability testing.

- NIST is reforming the FIPS certification process.