From IP ID to Device ID and KASLR Bypass

Amit Klein (joint research with Benny Pinkas)

Bar-Ilan University

Why do we need user (device) tracking?

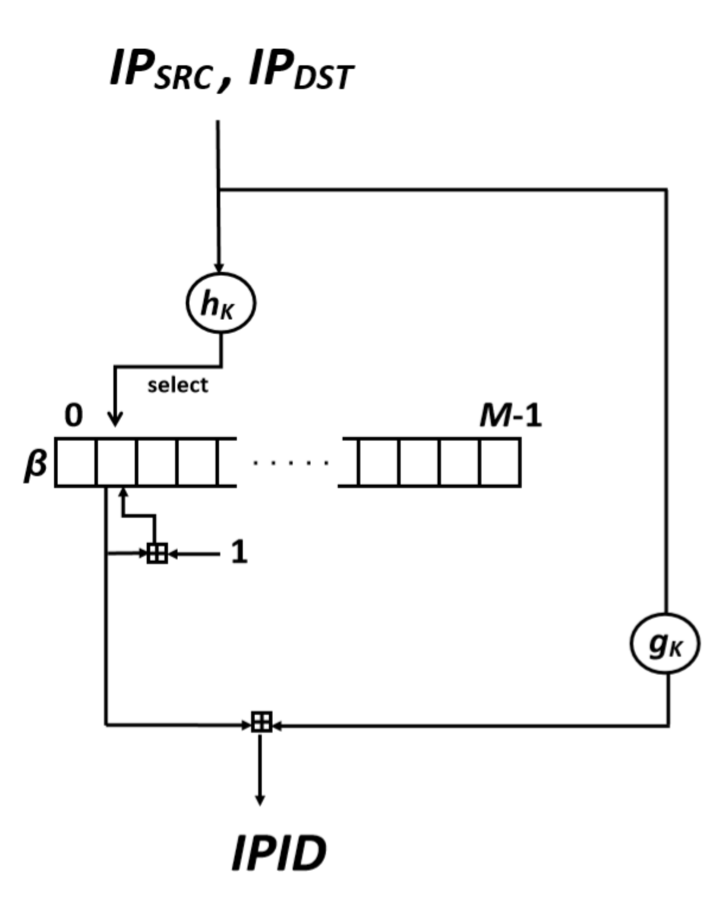
From the literature:

- **Real-time targeted marketing** (John Wilander, yesterday: “Cross Site Tracking”)
- **Campaign measurement**
- **Fraud detection**
- **Protection against account hijacking**
- Anti-bot and anti-scraping services
- Enterprise security management
- Protection against DDOS attacks
- Reaching customers across devices
- Limiting number of accesses to services

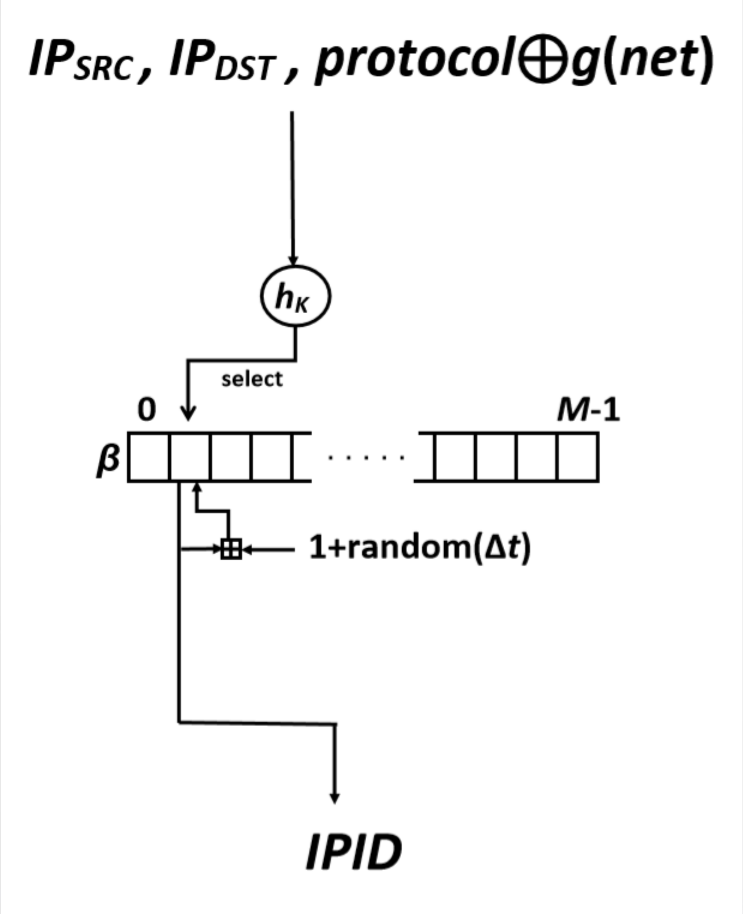
Introduction to IP ID

- IP ID – 16 bit IP header field
  - Identify fragments of the same IP datagram
  - Should not repeat "too closely" for same ‹IP_{SRC}, IP_{DST}, protocol›
  - Should **not** be predictable
- Implementation scheme (Windows, Linux+Android **stateless** protocols)
  - Large array of counters (*M*=2048/8192)
  - Hash function from ‹IP_{SRC}, IP_{DST}, protocol, **key**› to a counter
  - Increment the counter [Linux+Android: with extra randomness via t_{now}-t_{old}]
  - Use the result [Windows: add hash of ‹IP_{SRC}, IP_{DST}, $\mathbf{key_2}$›]

Windows



Linux



Attack setup

- Tracking HTML snippet, containing JS code
  - Can be embedded in any website
- The snippet forces the browser to connect to multiple attacker IPs
- Attacker collects IP ID for multiple (attacker) destination IPs
- We show how an attacker can calculate a **device** ID
  - Device ID remains unchanged across browsers, network switches, etc.
  - Can be used to track the user (device)
- Each snippet (site) can use a different set of destination IPs

Attack concept

- Based on cryptanalysis of the IP ID generation algorithm
- Requires IP IDs sent to multiple destinations (IP addresses)
- We use **collisions** of the hash values (array indices), which result in **related** counter values (same bucket, different times)

Attack concept

- We find the algorithm key (in full or in part) – 32 to 48 bits
  - This key is essentially unique per-device (up to the birthday paradox)

- The key is only regenerated at startup (Windows – only at **restart**):
  - Same key for all browsers, incl. privacy mode
  - Same key for all networks (incl. many VPNs!)
  - Invariant w.r.t. the set of destination IP addresses

Windows - The IP ID Algorithm

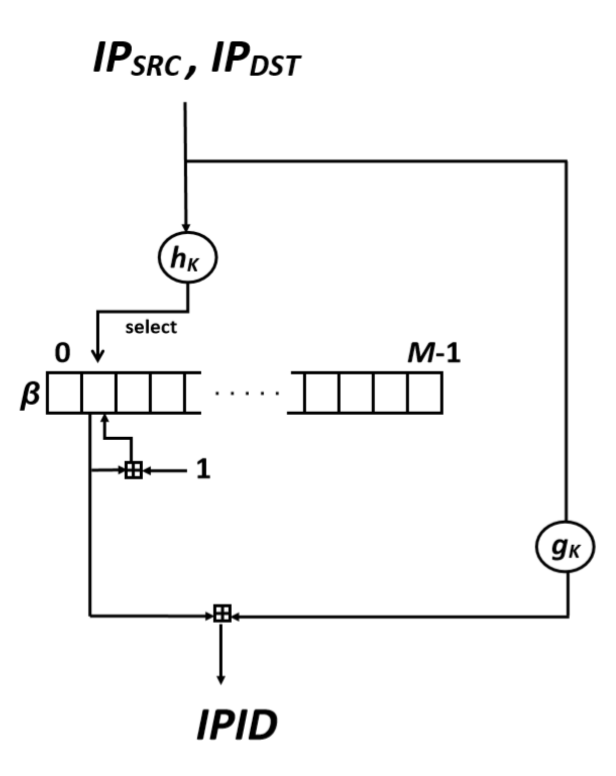
- β[] is the counter array, of size M=8192.
- IP ID generation algorithm (reverse engineered from tcpip.sys):

  $i \leftarrow h_{K,K2}(\text{class B of } IP_{DST}, IP_{SRC}) \bmod M$

  $v \leftarrow \beta[i] + (K1 \oplus T(K, IP_{DST} || IP_{SRC})) \bmod 2^{32}$

  $\beta[i]$++

  IPID $\leftarrow v \bmod 2^{15}$



- $K1$ (32 bits), $K2$ (32 bits), K (320 bits) - keys
- Hash function T (Toeplitz Hash) is bilinear (=very weak)

Windows Attack – Phase 1

- Note that the index *i* depends only on class B network of IP_{DST}
- Note that only 15 least significant bits of the counter β[*i*] are used
- Have several=6 IPs in the same class B, and obtain IP IDs for them:
  - All fall into the same counter β[*i*]
  - Enumerate over 2^{15} values of β[*i*], and get 15 linear equations over GF(2) on *K:*

    For IP_p and $IPID_p$, IP_q and $IPID_q$

    $IPID_x = \beta[i]+x+(K1 \oplus T(K, IP_x || IP_{SRC})) \bmod 2^{15}$

    $(IPID_p - \beta[i] - p) \oplus (IPID_q - \beta[i] - q) = T(K, IP_p || IP_{SRC}) \oplus T(K, IP_q || IP_{SRC})$

    $= T(K, IP_p \oplus IP_q)$
  - Solve linear equations to obtain 30 bits of *K* (16 high bits of $IP_p \oplus IP_q$ are 0)

Windows Attack – Phase 2

- Have several pairs of IPs, each pair in its own class B network
- Enumerate over additional 16 bits of *K*, to calculate any *T*(*K*,32-bit)

From phase 1:

$IPID_* = \beta[*] + (K1 \oplus T(K, IP_* || IP_{SRC} || 0^{32})) \bmod 2^{15}$

$K1 \oplus T(K, 0 || IP_{SRC} || 0^{32}) = (IPID_* - \beta[*]) \oplus T(K, IP_*) = \boldsymbol{X}$

- So (for each pair IP_0, IP_1 in the same class B network):

$IPID_j - j - (K1 \oplus T(K, IP_j || IP_{SRC} || 0^{32})) \bmod 2^{15} = \beta[\ldots]$

$IPID_j - j - (T(K, IP_j) \oplus \boldsymbol{X}) \bmod 2^{15} = \beta[\ldots]$

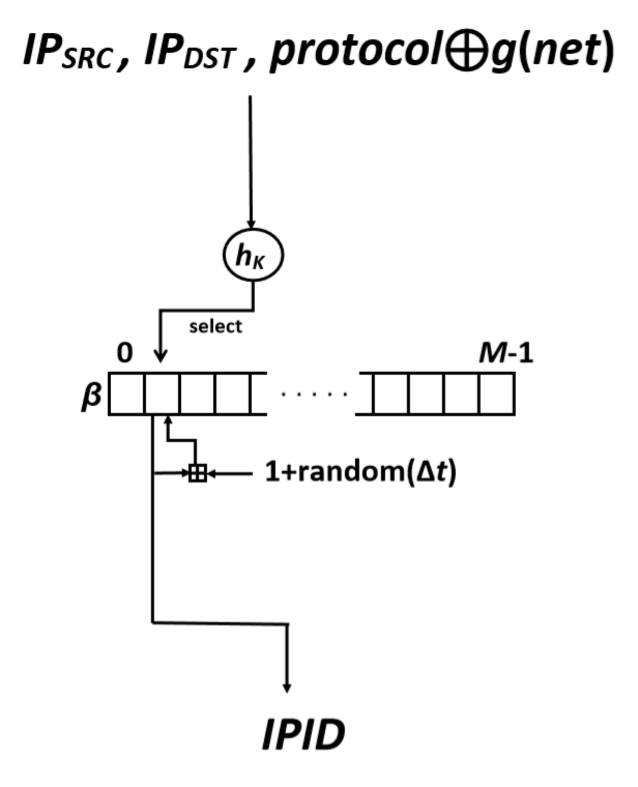
- Compare β[…] from *j*=0 and *j*=1, and eliminate

Linux+Android – Introduction to KASLR

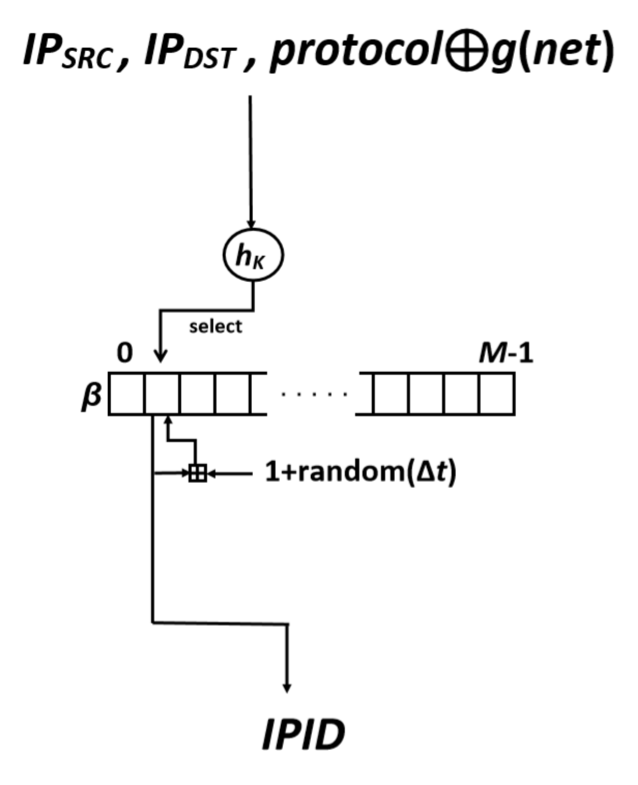
- KASLR=Kernel Address Space Layout Randomization
- ASLR is used to mitigate ROP (Return-Oriented Programming) and similar techniques
  - ROP is based on chaining ROP gadgets to form a (malicious) “program”
  - ROP gadget is code in a **known location**
  - ASLR randomizes the image load address (of modules, programs, etc.) to prevent the attacker from knowing the location of ROP gadgets
  - **KASLR** randomizes the kernel image load address. Enumeration is N/A since a “miss” results in O/S crash (very invasive…)
  - Typically KASLR adds a random offset (Linux – 9 bits, Android - 16 bits) in 2MB increments
- KASLR bypass = knowing kernel image address **offset**.

Linux+Android – stateless protocol (e.g. UDP) IP ID Algorithm

- Algorithm:
  - $i \leftarrow \mathbf{hash}_K(IP_{DST} || IP_{SRC} || protocol \oplus \mathbf{g}(net)) \bmod M$
  - $\beta[i] \leftarrow (\beta[i]+1+\mathbf{random}(\{0,\ldots,t_{now}-t[i]-1\})) \bmod 2^{16}$
  - $t[i] \leftarrow t_{now}$
  - $\mathbf{IPID} \leftarrow \beta[i]$
- M=2048, K is a 32 bit key, *protocol*=17 (UDP)
- t – in "jiffies" (100Hz/250Hz/300Hz) since boot



Linux+Android – stateless protocol (e.g. UDP) IP ID Algorithm



- Algorithm:

  $i \leftarrow \mathbf{hash}_K(IP_{DST}||IP_{SRC}||protocol \oplus \mathbf{g}(\mathbf{net}))$ mod M

  $\beta[i] \leftarrow (\beta[i]+1+\mathbf{random}(\{0,\ldots,t_{now}-t[i]-1\}))$ mod 2^{16}

  $t[i] \leftarrow t_{now}$

  IPID $\leftarrow \beta[i]$

- *M*=2048, *K* is a 32 bit key, *protocol*=17 (UDP)
- *t* – in “jiffies” (100Hz/250Hz/300Hz) since boot
- ***net*** – in kernel v4.1 and above, **kernel address** of net namespace struct (address publicly known per build, up to **KASLR offset**)
- **g**() – shift right by const (7/6/12) and truncate to 32 bits. Gets all the KASLR offset bits into the mix

The underlying issue in Linux/Android

Linux+Android Attack (simplified)

- Send a **burst** of *L*=400 UDP packets (one per IP address)
- Consider a bucket collision (same *i*) for two IP addresses:
  - A burst means that t_{now}-$t[i]$ is small and therefore **random**(0,…,t_{now}-$t[i]$-1) is small
  - Therefore, the 2nd packet IPID will be only slightly higher than the 1st packet IPID
  - Collect pairs of IP addresses that obey the above
  - There will be false positives
- Enumerate over a 32-bit key (for newer kernels – also the KASLR offset, 9-bit or 16-bit quantity)
  - For each key, count number of actual bucket collisions in the pairs collected
  - For a correct key this would be above some threshold (ν=11)
  - Enumeration is CPU intensive, may take time (esp. for 2^{48})
- **We also find the KASLR offset – hence KASLR bypass**

Vendor Status Following Our Reports

- Windows (**CVE-2019-0688**) – fixed by Microsoft in April 2019 Update
  - Nature of the fix – unknown. Presumably a different algorithm.
  - Undocumented registry setting can force fallback to the old (vulnerable) version ;-) (only for version<1903)
- Linux
  - KASLR bypass (**CVE-2019-10639**) – fixed mainline (5.1-rc4), stable (5.0.8) and all relevant long term versions (4.19.35, 4.14.112, 4.9.169, 4.4.179)
  - Also extends key size to 64-bit
  - Extend key size to 64-bit in 3.18.139, 3.16.67 via a patch contributed by the authors
  - Switch to SipHash and 128-bit key (**CVE-2019-10638**) – 5.2-rc1, 5.1.7, 5.0.21, 4.19.48, 4.14.124 (+ 3.16.72 released August 13th)

Conclusions

- Security/privacy is a concern, even when generating seemingly non-security data
- Use industrial-strength crypto
- Use adequate-sized key
- Don’t use sensitive data as key

Q&A

Thanks!

Extended version of the paper:

https://arxiv.org/pdf/1906.10478.pdf