Filesystems timing attacks

ZeroNights, Moscow, 08/11/13
Timing attacks basics

time to execution of
Function(UserData, PrivateData)
depends from UserData and PrivateData

dthis time can be use to determine PrivateData by UserData
Filesystems timing attacks

What is Function(UserData,PrivateData) ?

Basically - STAT, but not only
FS timing attacks intro

execution time of search operation depends on:

- search string
- data on which searches for

attack concept is determine data by timings on different search strings
FS timing attacks intro

execution time of search operation depends on:

- search string
- data on which searches for

attack concept is determine data by timings on different search strings
Directory indexing mechanism
● list
● BTree (not binary tree)
● HTree

+ cache mechanism
<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Directory indexing algo</th>
<th>Hash type</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext2</td>
<td>list</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>ext3/4</td>
<td>htree</td>
<td>half_md4 + seed (earlier Legacy, TEA)</td>
<td>+</td>
</tr>
<tr>
<td>ufs2/NFS</td>
<td>dirhash</td>
<td>FNV (FreeBSD) DJB (OpenBSD)</td>
<td>+</td>
</tr>
<tr>
<td>FAT</td>
<td>list (btree)</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>NTFS</td>
<td>btree</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
To cache or not to cache

- Cache does not prevent timing attacks
- Cache remove disk operations noises
ext2 lists

To find a file, the directory is searched front-to-back for the associated filename

HTree indexes were originally developed for ext2 but the patch never made it to the official branch. The dir_index feature can be enabled when creating an ext2 filesystem, but the ext2 code won't act on it.
ext2 lists

./fs/ext2/dir.c:

static inline int ext2_match (int len, const char * const name,
                         struct ext2_dir_entry_2 * de)
{
    if (len != de->name_len)
        return 0;
    if (!de->inode)
        return 0;
    return !memcmp(name, de->name, len);
}

Timing anomaly for files with unexisting length
ext2 results

10 loops
100k STATS/loop

Time(compared bytes)
OPTIMIZATION
Returns the hash of a filename. If len is 0 and name is NULL, then this function can be used to test whether or not a hash version is supported.

The seed is an 4 longword (32 bits) "secret" which can be used to uniquify a hash. If the seed is all zero's, then some default seed may be used.
ext3/4 HTree

./fs/ext3/hash.c: ext3fs_dirhash
* Returns the hash of a filename. If len is 0 and name is NULL, then
* this function can be used to test whether or not a hash version is
* supported.
*
* The seed is an 4 longword (32 bits) "secret" which can be used to
* uniquify a hash. If the seed is all zero's, then some default seed
* may be used.

4x32 bites = 16 bytes
- impossible to brute force ;( 
ext3/4 predicted seed

- Useful while filesystem comes from firmware image
- All devices with same firmwares have the same seeds
What hash type used ext3/4?

man tune2fs

```bash
hash_alg=hash-alg
```

Set the default hash algorithm used for filesystems with hashed b-tree directories. Valid algorithms accepted are: legacy, half_md4, and tea.

half_md4 by default
ext3/4 MD4 hash tricks

\[
p = \text{name};\]
\[
\text{while } (\text{len} > 0) \{
    (*\text{str2hashbuf})(p, \text{len}, \text{in}, 8);
    \text{half_md4_transform}((\text{buf}, \text{in}));
    \text{len} -= 32;
    p += 32;
\}
\]

\[
\text{minor\_hash} = \text{buf}[2];\]
\[
\text{hash} = \text{buf}[1];\]
\[
\text{break};\]
ext3/4 MD4 hash tricks

MD4($salt.$filename) - really?
If you know MD4($salt."a")
You know MD4($salt."a".$postfix)
W/o knowledge about $salt value!

What is $salt?
Seed which unique for whole current filesystem (all folders)
ext3/4 legacy hash

static __u32 dx_hack_hash_signed(const char *name, int len)
{
    __u32 hash, hash0 = 0x12a3fe2d, hash1 = 0x37abe8f9;
    const signed char *scp = (const signed char *) name;
    while (len--) {
        hash = hash1 + (hash0 ^ (((int) *scp++) * 7152373));
        if (hash & 0x80000000)
            hash -= 0x7fffffff;
        hash1 = hash0;
        hash0 = hash;
    }
    return hash0 << 1;
}
Binary search for timing attack

ext3_find_entry -> ext3_dx_find_entry -> dx_probe:

\[ p = \text{entries} + 1; \]
\[ q = \text{entries} + \text{count} - 1; \]
while (p <= q)
{
    \[ m = p + (q - p)/2; \]
    \[ \text{dxtrace(printk(".")};) \]
    if (dx_get_hash(m) > hash)
        \[ q = m - 1; \]
    else
        \[ p = m + 1; \]
}
ufs2/NFS FNV hash - no seed/salt!

static __inline Fnv32_t
fnv_32_buf(const void *buf, size_t len, Fnv32_t hval)
{
    const u_int8_t *s = (const u_int8_t *)buf;

    while (len-- != 0) {
        hval *= FNV_32_PRIME;
        hval ^= *s++;
    }
    return hval;
}
ufs2/NFS DJB hash - no seed/salt!

#define HASHINIT 5381
#define HASHSTEP(x,c) (((x << 5) + x) + (c))

hash32_buf(const void *buf, size_t len, uint32_t hash)
{
    const unsigned char *p = buf;
    while (len--)
    {
        hash = HASHSTEP(hash, *p++);
    }
    return hash;
}
UFS search by filename

ufs_lookup -> ufs_lookup_ino:

    switch (ufsdirhash_lookup(dp, cnptr, &i_offset, &bp, nameiop == DELETE ? &prevoff : NULL)) {
        case 0:
            ep = (struct direct *)((char *)bp->b_data + (i_offset & bmask));
            goto foundentry;
        case ENOENT:
            i_offset = roundup2(dp->i_size, DIRBLKSIZ);
            goto notfound;
        default: break;
    }

ufsdirhash_lookup:

    ... for (; (offset = DH_ENTRY(dh, slot)) != DIRHASH_EMPTY;
        slot = WRAPINCR(slot, dh->dh_hlen)) {
        ... if (dp->d_namlen == namelen &&
            bcmp(dp->d_name, name, namelen) == 0) {
            /* Found. Get the prev offset if needed. */
            if (prevoffp != NULL) {
                if (offset & (DIRBLKSIZ - 1)) {
                    prevoff = ufsdirhash_getprev(dp, offset);
                    if (prevoff == -1) {
                        error = EJUSTRETURN;
                        goto fail;
                    }
                } else
                ...
FAT/NTFS results

- BTree + binary search - no hashes, no problems ;)
- Just test using PoC from github
PoC

● Simple tool that can demonstrate timing anomaly
● Just PoC, not a framework
● Framework soon ;)

https://github.com/wallarm/researches/blob/master/fs-timing/fs-timing.c
Remote attacks

- Network noises
- Lack of opportunity to request multiple files in same loop
- But you can use additional features:
  - CPU overload
  - inodes count
  - memory usage

I think you know how to do it remotely ;)

Real case from a wild

- TFTP service
- Classic bruteforce w/o results
- Times to retrieve files are different
- Sort it!
- Find prefixes with anomaly timings:
  - rom-
  - firmware.
  - ...
- Brute filename after prefixes
Next steps

- And... YES!
- We want to optimize classic DirBusting technology
- For brute force to search through timing-attacks!
The end

Contacts:

@wallarm, @d0znpp

http://github.com/wallarm

no+SQL timing attacks at: