

# Filesystems timing attacks

ZeroNights, Moscow, 08/11/13



# Timing attacks basics

**time** to execution of

**Function(UserData, PrivateData)**

depends from UserData and PrivateData

this **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



The diagram consists of three arrows. One arrow starts from the word 'search' in the second bullet point and points to the word 'search' in the attack concept text. A second arrow starts from the word 'data' in the second bullet point and points to the word 'timings' in the attack concept text. A third arrow starts from the word 'timings' in the attack concept text and points back to the word 'time' in the first sentence.

# Filesystems search basics

Directory indexing mechanism

- list
- BTree (not binary tree)
- HTree

+ cache mechanism

<b>Filesystem</b>	<b>Directory indexing algo</b>	<b>Hash type</b>	<b>Cache</b>
ext2	list	-	+
ext3/4	htree	half_md4 + seed (earlier Legacy, TEA)	+
ufs2/NFS	dirhash	FNV (FreeBSD) DJB (OpenBSD)	+
FAT	list (btree)	-	+
NTFS	btree	-	+

# 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);  
}
```



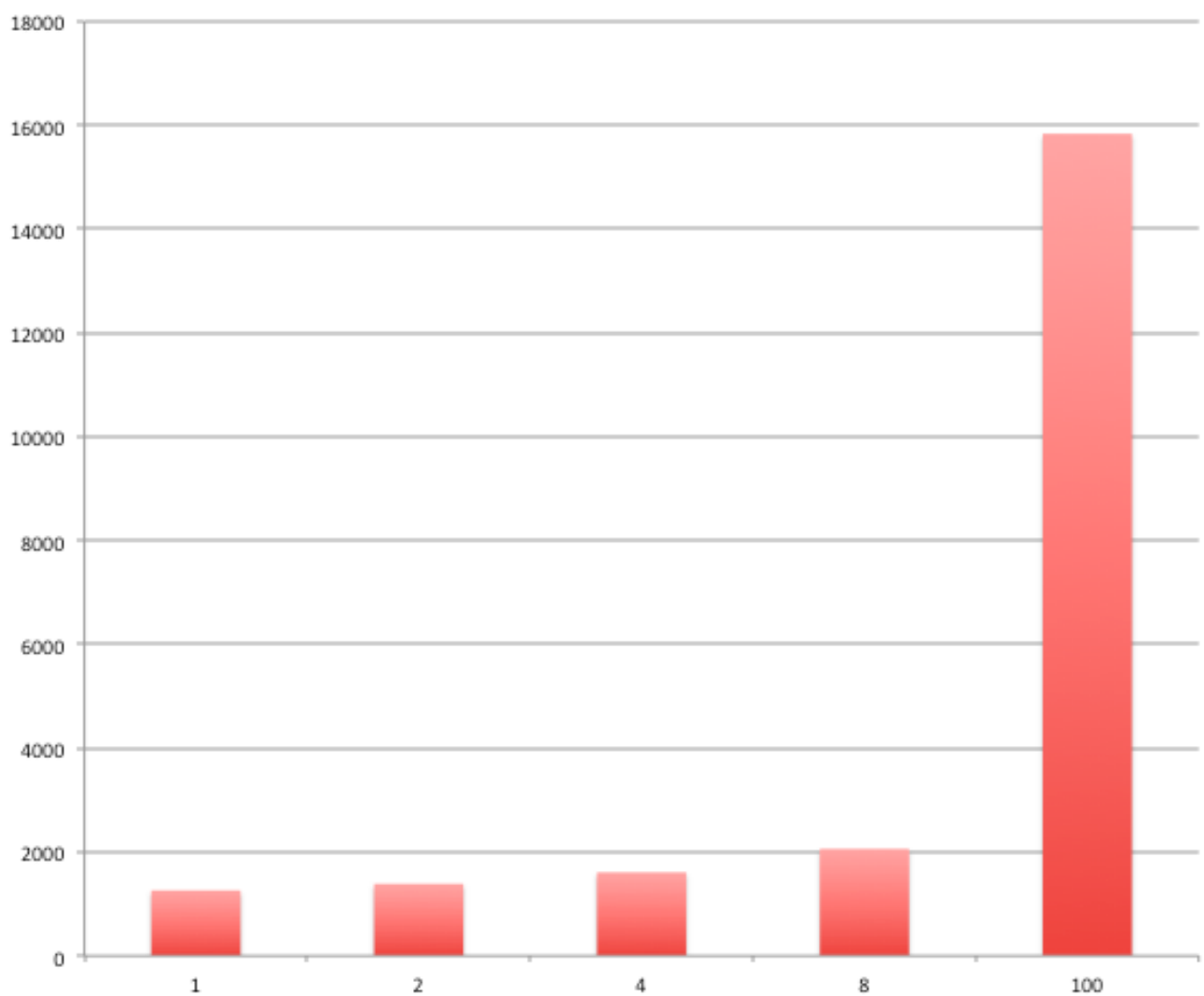
Timing anomaly for  
files with unexisting  
length

# ext2 results

10 loops

100k STATS/loop

Time(compared bytes)





**OPTIMIZATION**



# 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
- \* unquify a hash. If the seed is all zero's, then some default seed
- \* may be used.

# ext3/4 HTree

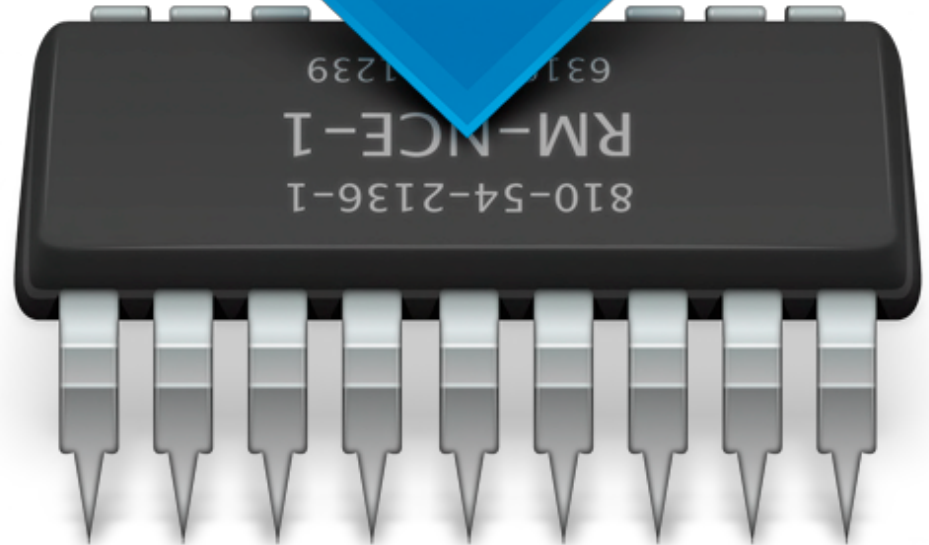
4x32 bites = 16 bytes  
- impossible to brute  
force ;(

./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
- \* unify a hash. If the seed is all zero's, then some default seed
- \* may be used.

# ext3/4 predicted seed

- Usefull while filesystem comes from **firmware** image
- All devices with same firmwares has the **same seeds**



# What hash type used ext3/4 ?

man tune2fs

hash\_alg=hash-arg

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 = name;
while (len > 0) {
    (*str2hashbuf)(p, len, in, 8);
    half_md4_transform(buf, in);
    len -= 32;
    p += 32;
}
minor_hash = buf[2];
hash = buf[1];
break;
```

mkfs.ext3/4  
seed from  
/dev/urandom

seed, 16 bytes  
(4x32 bits long)  
“secret”

128 bit of state  
require to calculate  
next hashes

# 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;
}
```

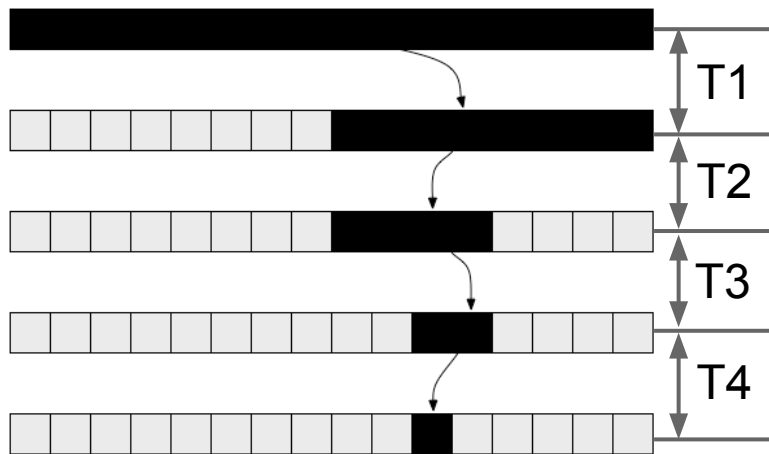
# Binary search for timing attack

ext3\_find\_entry -> ext3\_dx\_find\_entry -> dx\_probe:

```
p = entries + 1;
q = entries + count - 1;
while (p <= q)
{
    m = p + (q - p)/2;
    dxtrace(printf("."));
    if (dx_get_hash(m) > hash)
        q = m - 1;
    else
        p = m + 1;
}
```

1.  $\text{min\_hash} \leq \text{hash} \leq \text{max\_hash}$
2.  $(\text{max} - \text{min})/2 \leq \text{hash}$
3. ...

$$T = T1 + T2 + T3 + T4$$



# 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;
}
```



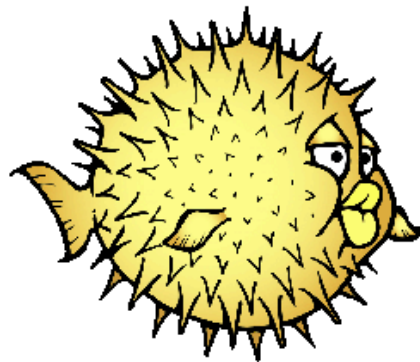
FreeBSD

# 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;
}
```



**OpenBSD**

# UFS search by filename

```
ufs_lookup -> ufs_lookup_ino:
```

```
switch (ufsdirehash_lookup(dp, cnp->cn_nameptr, cnp->cn_namelen,
    &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;
```

```
ufsdirehash_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 = ufsdirehash_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 bruteforce to search through timing-attacks!

# The end

Contacts:

@wallarm, @d0znpp

<http://github.com/wallarm>

no+SQL timing attacks at:

