Exploiting Samba

Easy as PIE…

Compass Security AG
Werkstrasse 20
Postfach 2038
CH-8645 Jona

Tel +41 55 214 41 60
Fax +41 55 214 41 61
team@csnc.ch
www.csnc.ch

Presentation Overview

- The Security Announcement
- The Vulnerability
- The Heap
- Basic Exploit Development
- Advanced Exploit Development
- The Cake is a lie

The Security Announcement

Or: ZDI ftw

A wild CVE appears…

- CVE-2012-1182
- From 10 April 2012
- Samba 3.0.x - 3.6.3 (inclusive)
- Remote Root Code Execution
- Unauthenticated

As this does not require an authenticated connection it is the most serious vulnerability possible in a program, and users and vendors are encouraged to patch their Samba installations immediately.

The Security Announcement

CVE-ID

CVE-2012-1182
(under review)

Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

The RPC code generator in Samba 3.x before 3.4.16, 3.5.x before 3.5.14, and 3.6.x before 3.6.4 does not implement validation of an array length in a manner consistent with validation of array memory allocation, which allows remote attackers to execute arbitrary code via a crafted RPC call.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- CONFIRM:https://www.samba.org/samba/security/CVE-2012-1182

Status

Candidate

This CVE Identifier has "Candidate" status and must be reviewed and accepted by the CVE Editorial Board before it can be updated to official "Entry" status on t
rejected in the future.

Not much data in CVE...

Phase

Assigned (20120214)

Votes

Comments

Candidate assigned on 20120214 and proposed on N/A

SEARCH CVE USING KEYWORDS: Submit

You can also search by reference using the CVE Reference Maps.

FOR MORE INFORMATION: cve@mitre.org

The Vulnerability – finding it

Follow the right people..

- And get it while it's hot
- 3 archives with about 12 different PoCs…

The Vulnerability

Or: WTF?

The Vulnerability – analyze it

In reproducers0.zdi.tar/zdi/ZDI-CAN-1504.poc

```
[1] ndr_pull_uint32(ndr, NDR_SCALARS, &r->count);
[2] ndr_pull_array_size(ndr, &r->settings));
[3] NDR_PULL_ALLOC_N(ndr, r->settings, ndr_get_array_size(ndr, &r->settings));
[4] for (cntr_settings_1 = 0; cntr_settings_1 < r->count; cntr_settings_1++) {
     ndr_pull_lsa_PolicyAuditPolicy(
                    ndr,
                    NDR_SCALARS,
                    &r->settings[cntr_settings_1]));
    }

We control the value of r->count (line #1). Also we control the size of array (line #2).
On line #3 memory is allocated for r->settings array. On line #4 we overflow r->settings
if we set r->count > array_size

static enum ndr_err_code
ndr_pull_lsa_PolicyAuditPolicy(
            struct ndr_pull *ndr,
            int ndr_flags,
            enum lsa_PolicyAuditPolicy *r)
{
      uint32_t v;
      NDR_CHECK(ndr_pull_enum_uint32(ndr, NDR_SCALARS, &v));
      *r = v;
      return NDR_ERR_SUCCESS;
}
```

The Vulnerability – analyze it

Pseudocode:

```
r->count = packet.count
r->settings = alloc(packet.settingsLen)

for (int n=0; n < r->count; n++) {
   settings[n] = getNextByteFromPacket();
}
```

Values from PoC:

- count = 4096
- settingsLen = 100
- length of packet: 10’000 Bytes

The Vulnerability – analyze sorroundings

- What can we overwrite?
- The Heap!
  - Because destination buffer is malloc'ed
  - Lets read the article about jemalloc from phrack (default allocator in FreeBSD)!
  - Some hours later…
  - Find out Samba uses ist own allocator (talloc)…

Interlude

- Remote buffer overflow in samba
- Size of destination buffer and data can be different
  - And are specified by client
- Able to overwrite Heap Data

The Heap

Or: talloc ftw!

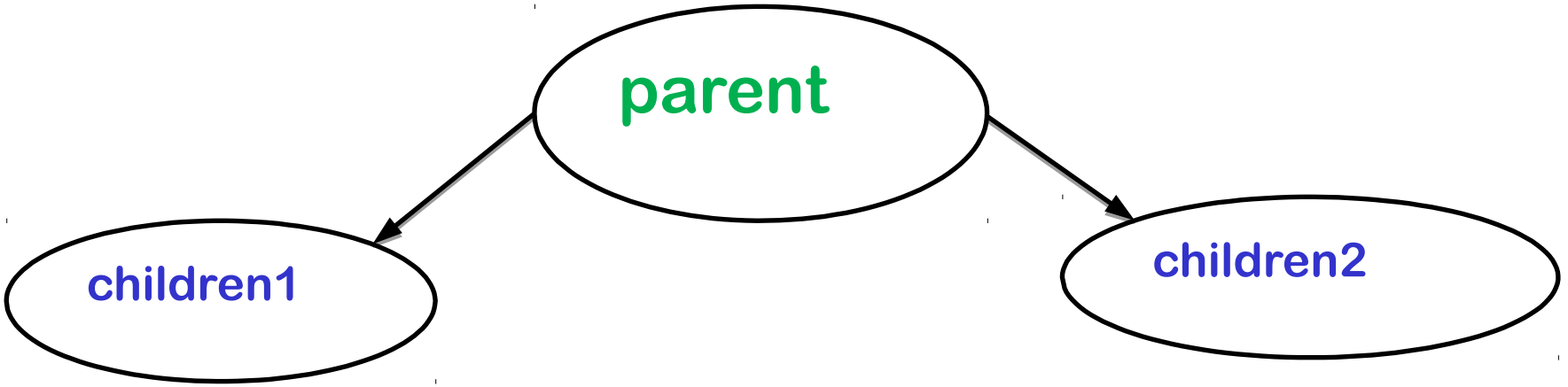
Heap allocator: talloc - introduction

- «Tree» Allocator
  - A Hierarchical Allocator
- Every talloc-returned pointer is a pool by itself
- Can free children
- Or free parent
  - Which in turn, frees all children of parent
- Supports destructors

Heap allocator: talloc - usage

```
*parent = talloc(NULL, struct a)
*children1 = talloc(parent, struct a)
*children2 = talloc(parent, struct a)

talloc_free(children1); // frees children1
talloc_free(parent);    // frees parent & children2
```



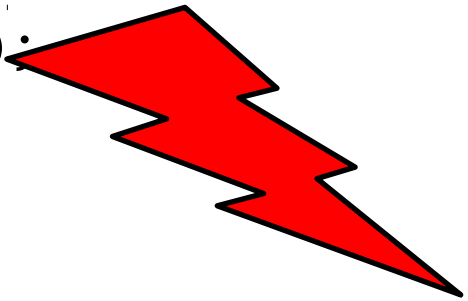
Heap allocator: talloc – programming

```
*listentry = talloc(NULL, struct a);
list_add(&global_list, listentry);



talloc_free(listentry);


list_get(&global_list, &aListentry );
```



Use after free()

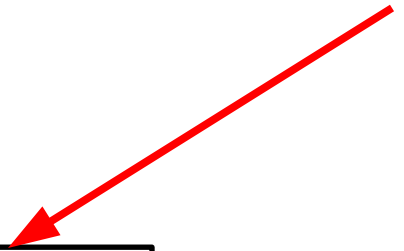
Heap allocator: talloc - destructors

```
*listentry = talloc(NULL, struct a);
list_add(&global_list, listentry);

talloc_set_destructor(listentry, parent_destroy);

talloc_free(listentry);    // calls destructor


list_get(&global_list, &aListentry);
```

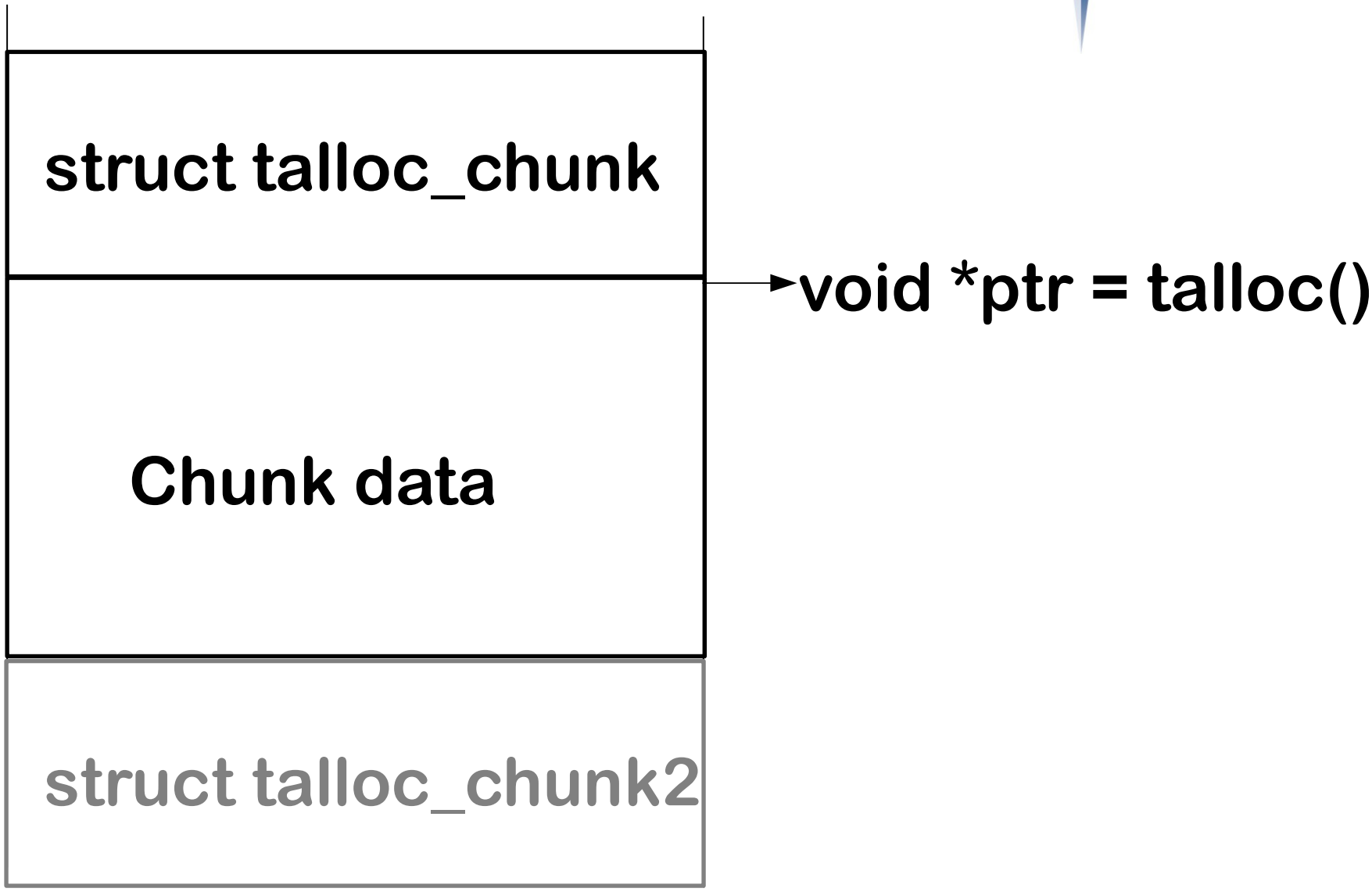


```
static int parent_destroy(struct a *ptr) {
    list_del(&global_list, ptr);
}
```

Heap allocator: talloc – data structure

```
struct talloc_chunk {
    struct talloc_chunk *next, *prev;
    struct talloc_chunk *parent, *child;
    struct talloc_reference_handle *refs;
    talloc_destructor_t destructor;
    const char *name;
    size_t size;
    unsigned flags;
    void *pool;
};
```

Heap alloctor: talloc memory layout



Exploit Development

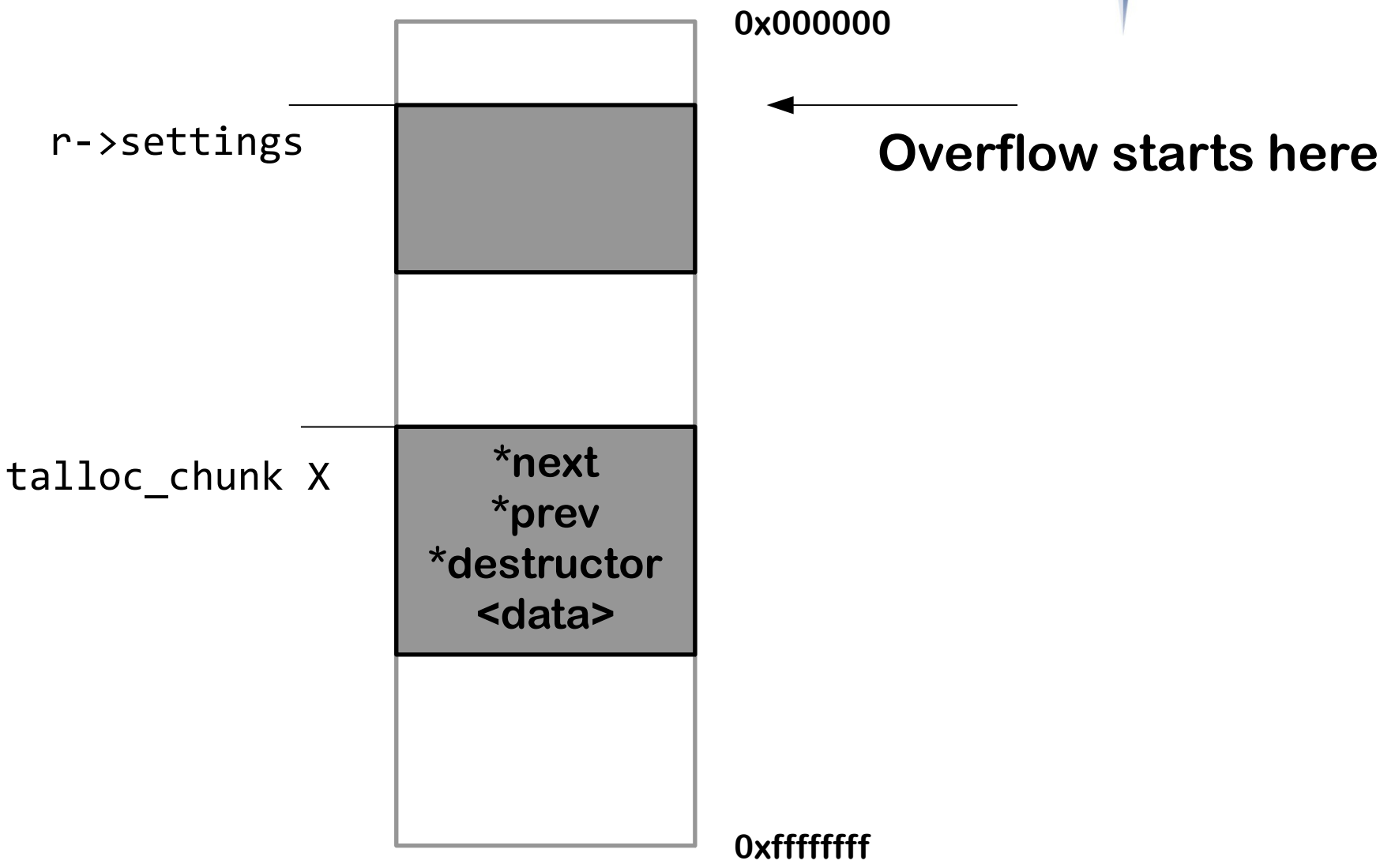
Exploit development - overview

1. Overflow heap with A's
2. Look in gdb where it crashes
3. Calculate amount of bytes between overflow location and chunk
4. Create a special chunk at that location with our destructor

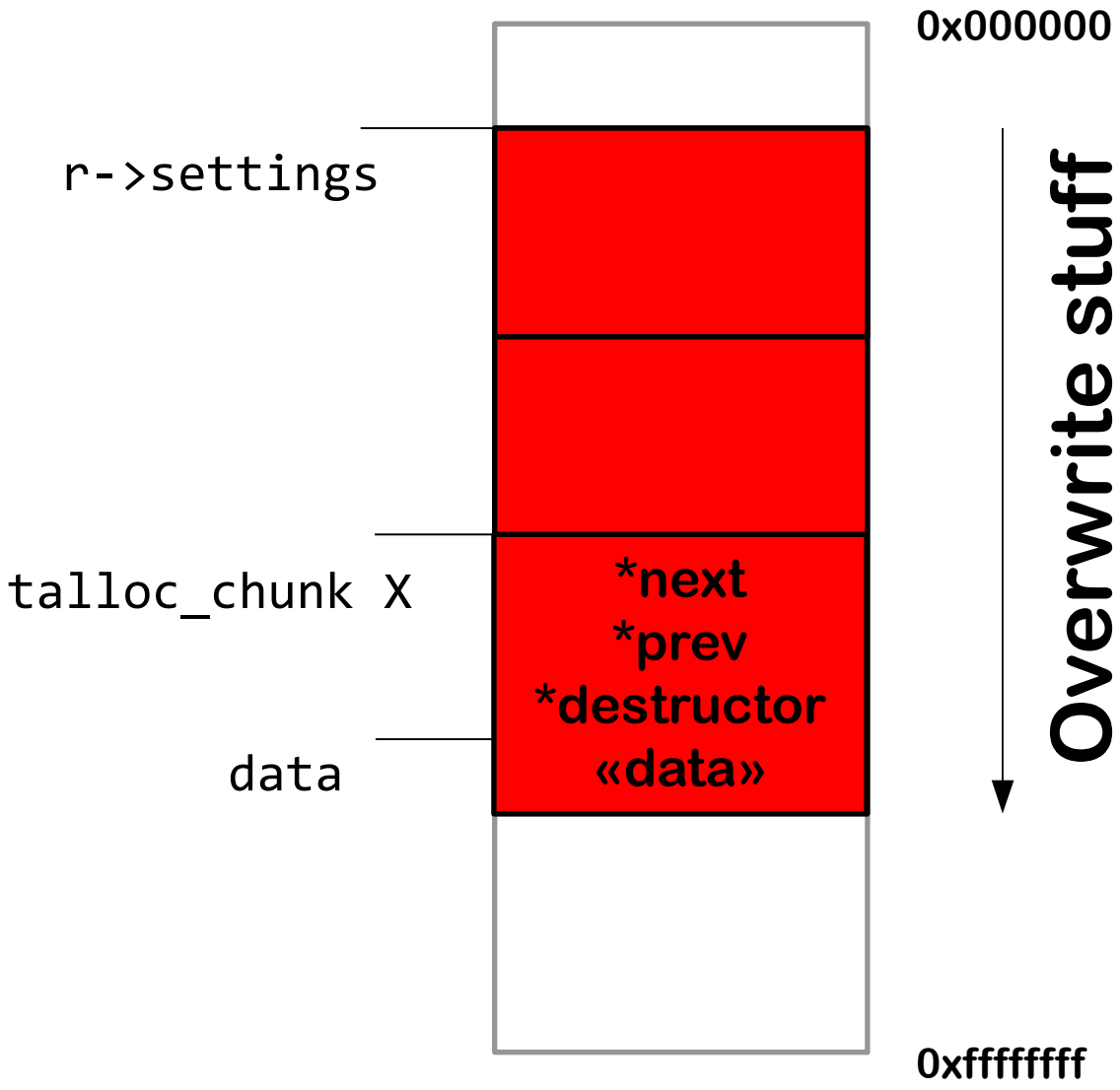
Exploit development - overview

1. Overflow heap with A's
   - With constraints (array size, count size, amount of bytes in packet..)
2. Look in gdb where it crashes
   - should crash in talloc_free()
   - But not on call(), but when accessing *parent or magic
   - Identify start of chunk
3. Calculate amount of bytes between overflow location and chunk
   -
4. Create a special chunk at that location with our destructor
   - So on talloc_free(), the destructor gets called
   - set magic bytes so the chunk passes all talloc_free() checks

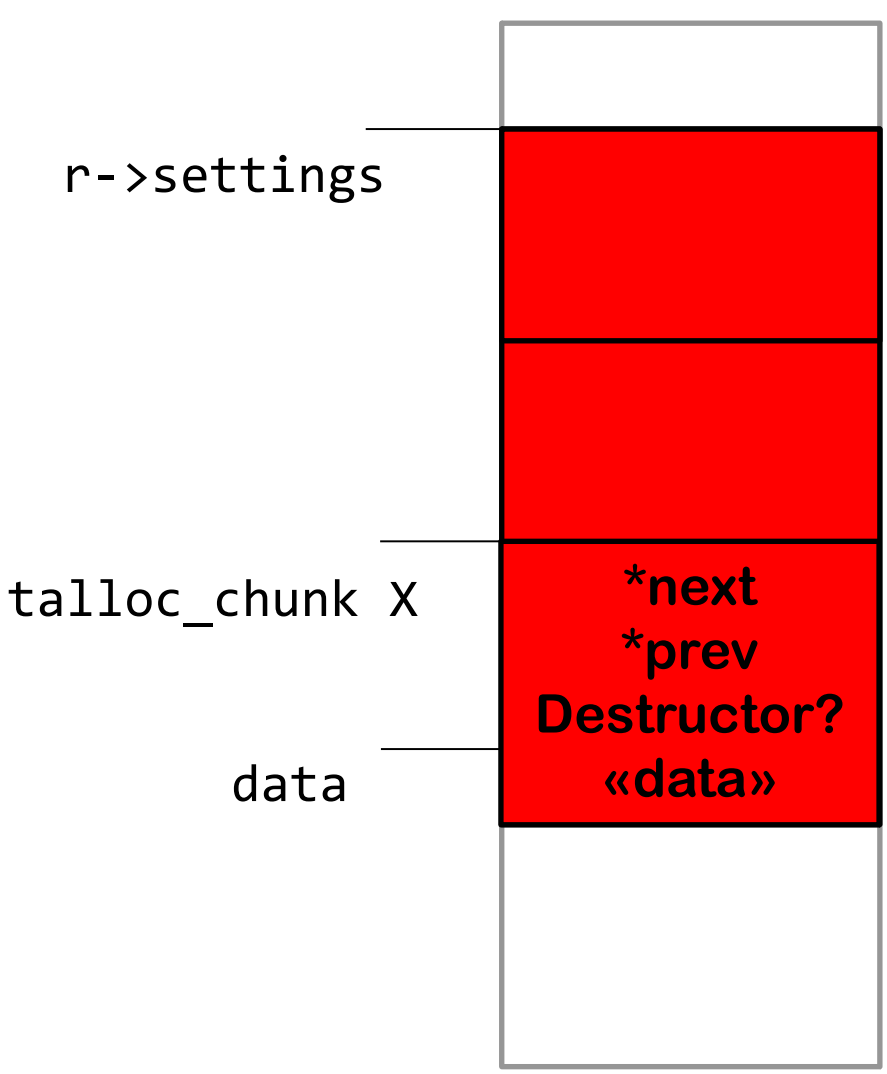
Exploit development – initial memory



Exploit development – overflowed memory



Exploit development – code execution



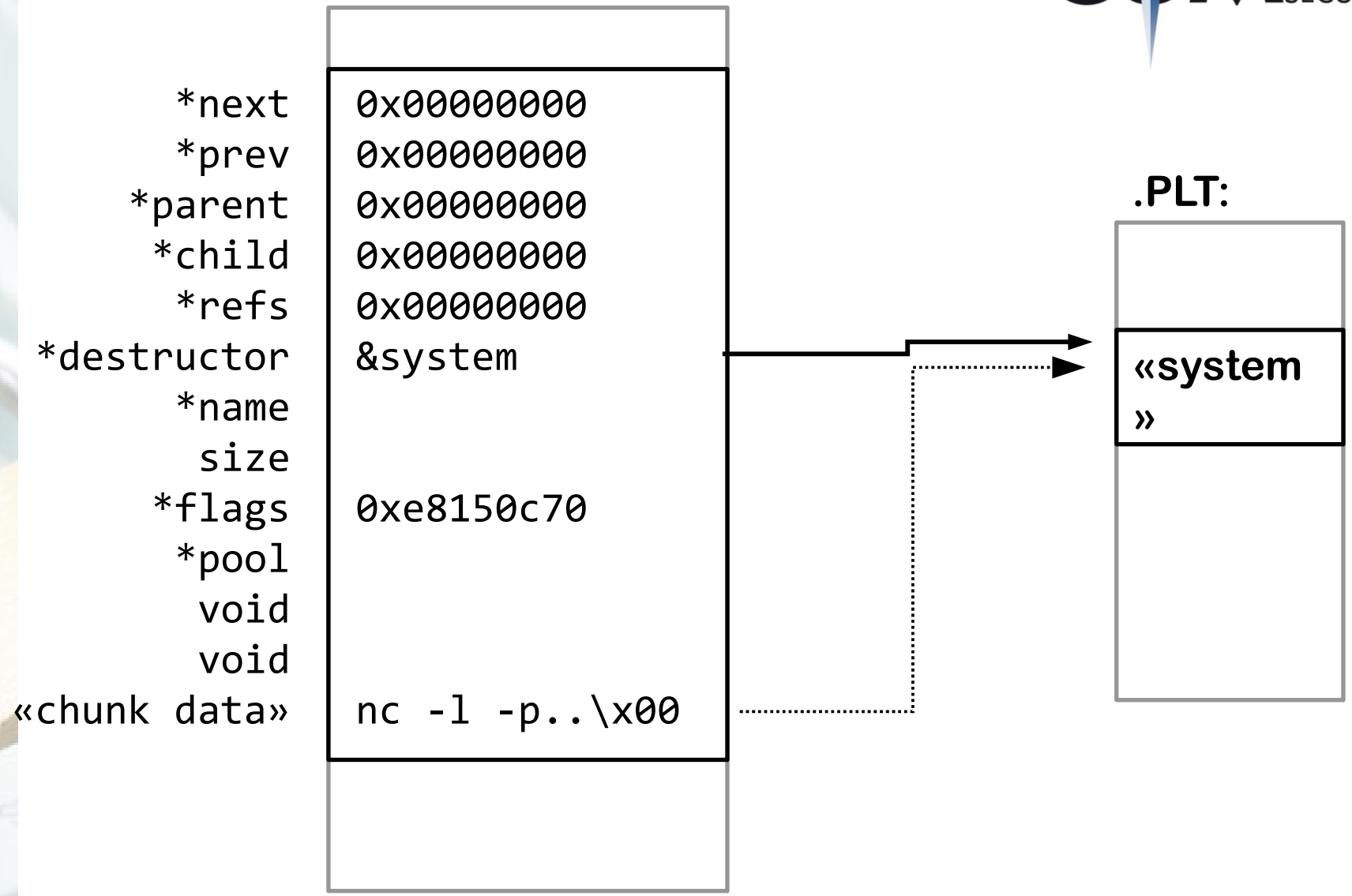

```
call
  chunkX.destructor(&data);
```

```
talloc_free(data);
```

Exploit development

- What to call?
- destructor(&data)
  - We've seen that before!
  - system(&data)
  - Data = «touch /tmp/bla»

Exploit development



Interlude

1. Find chunk which get's free()'d
2. Overwrite destructor with system()
3. Put Shellcode into data of chunk
4. Profit!

Common security technologys:

- NX:
  - Cant upload our own code ☹
  - cant fill heap with NOP sled ☹
  - Do not need for ret2libc ☺
- ASLR: stack and heap adresses are randomized
  - Can't use static heap or stack adressess ☹
  - Do not need: talloc will provide pointer to our shellcode ☺
- PIE: Code and PLT are randomized
  - Darn, we cant jump back to system@plt in libc! ☹
  - Or can we? ☺

Exploit development - PIE

Samba will be compiled as PIE

Makefile:

```
PICFLAG     = -fPIC
LIBS        = -lresolv -lresolv -lnsl -ldl -lrt
LDFLAGS     = -pie -Wl,-z,relro -L./bin
DYNEXP      = -Wl,--export-dynamic
LDSHFLAGS   = -fPIC -shared -Wl,-Bsymbolic
              -Wl,-z,relro -L./bin -lc -Wl,-z,defs
SHLIBEXT    = so
SONAMEFLAG = -Wl,-soname=
```

Exploit development - PIE

Adress of system() in binarys compiled with PIE?

BackTrack 5R1:

- 0xb7???100

Ubuntu 11.10:

- 0x00???b20

Can we brute force it?

- 16^3 = **4096** = 12 Bits of Entropy
- PIE base address changes only on exec()
  - not on fork()
  - ☺

Exploit development - PIE

```
Addr: 0x000b20
Addr: 0x001b20
Addr: 0x002b20
Addr: 0x003b20
Addr: 0x004b20
Addr: 0x005b20
Addr: 0x006b20
Addr: 0x007b20
Addr: 0x008b20
Addr: 0x009b20
Addr: 0x00ab20
Addr: 0x00bb20
[…]
Addr: 0xfffb20
```

```
n = 0x000b20
for x in range (0, 0x1000)
   addr = n + (x * 0x1000)
   exploit(n)
```

Demo

<<Demo>>

Randomness


```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

How random is random?

Test Randomness of PLT:

```
root@bt:~# cat test.c
#include <stdio.h>
#include <stdlib.h>

void main(void) {
    printf("%p\n", &system);
}

root@bt:~# gcc -pie test.c -o pie
root@bt:~# while true; do ./pie  >> addr.txt; done
^C
```

How random is random?

- Test on BackTrack
- Generate 650'000 processes

```
$ cat addr.txt | sort | uniq –c | sort -n

        1172 0xb76c6100
        1182 0xb7713100
        […]
        1378 0xb77a0100
        1382 0xb7633100
        1386 0xb779d100
        2551 0xb76af100

        511 Different!
```

Summary

OS	# of Addresses	Deviation	Rating
BackTrack 5.0R1	511	2x	FAIL
Ubuntu 10.11	3445	150x	FAIL
CentOS 5.8 Final	3480	300x	FAIL
FreeBSD 8.2	1	-	-
BackTrack 5R2 64	>1'000'000	1	PASS
OpenBSD –current	65421	18x	PASS

How to compile

What compile option to choose?

–pie, -fpie, -fPIE, -fpic, -fPIC

Correct:

-pie