suresec
SECURING THE SOURCE

Exploiting OpenBSD

Ben Hawkes

ben@suresec.com.au

Introduction

- Stack Smashing Protection (SSP/ProPolice)
- Address Space Layout Randomization (ASLR)
- OpenBSD’s custom memory allocator
- Some other points of interest

Note: this presentation is primarily focused on the i386 port of OpenBSD 3.9

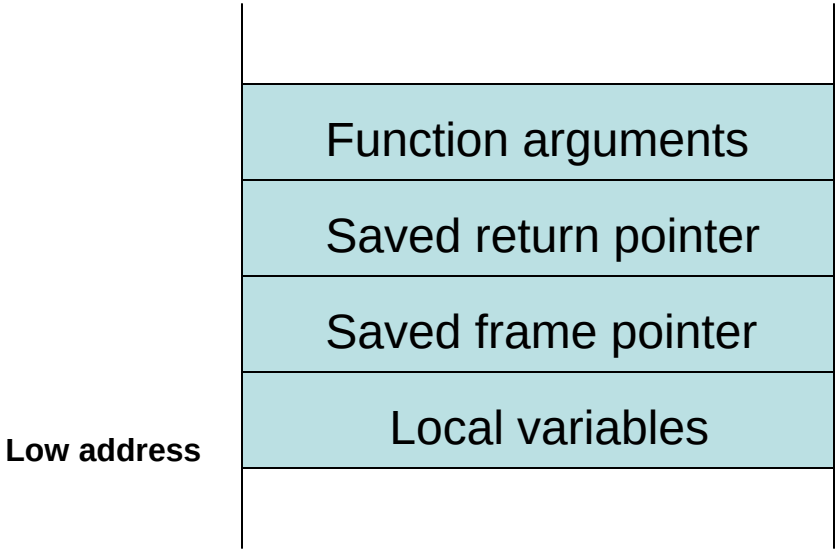
SSP/ProPolice

Stack Smashing Protector adopted by OpenBSD to provide stack frame canaries

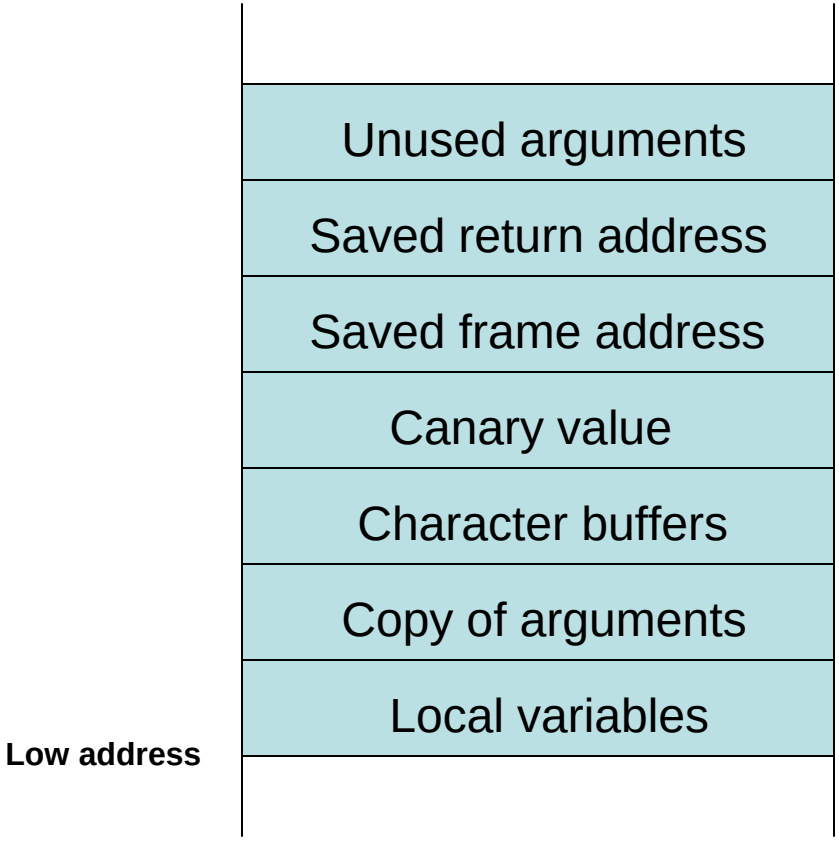
SSP also rearranges stack frame to mitigate linear overflow damage

Canary is usually 32-bit long random number (arc4random in OpenBSD)

Classic Stack Frame Layout



SSP Stack Frame Layout



SSP/ProPolice

In order to overflow return address, must have knowledge of canary

Canary is created at runtime by libc constructor function (__guard_setup)

Canary is randomized whenever libc is loaded

Effectively every time execve( ) is used, but significantly not when fork( ) is used

SSP/ProPolice

How can attacker “hit” the right canary value?

One method is to find an information leak in same application as overflow

Arbitrary read on stack or binary’s data section may give attacker knowledge of canary value

Only possible when leak and overflow both occur before libc is reloaded

How to identify canary value? Kolmogorov complexity, address space heuristics, or both

SSP/ProPolice

Finding a useable information leak is unlikely

Need a more generic technique

Perhaps the most generic technique is brute force

But… 2^32 = 4 billion different canary values

Which gives an average of 2 billion attempts before success

Not feasible to do this remotely, barely possible to do locally

SSP/ProPolice

Introducing “byte for byte” brute forcing

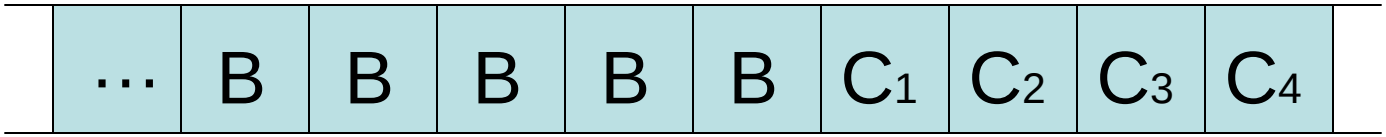
Some vulnerabilities can be brute forced in an average of 512 attempts

Only possible when canary stays same between crashes (i.e. forking daemon)

And only when the overflow is not null terminated (potentially from read, memcpy, strncpy, loops)

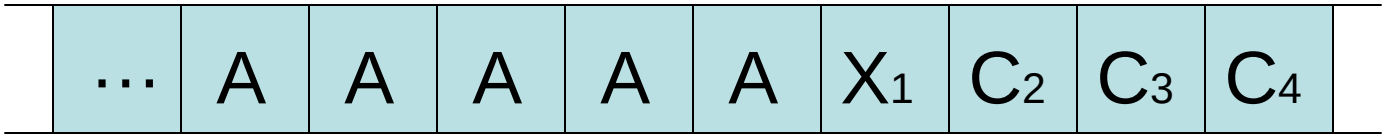
Technique is to brute force each byte of the canary individually along with time analysis

SSP/ProPolice



C_n = 4 canary bytes (total 32-bit, dword)
B = buffer bytes (assuming no padding)

SSP/ProPolice



A = buffer overflow padding

X_1 = brute force byte, from 0 to 255

SSP/ProPolice

How can you tell when the brute force byte equals the canary byte?

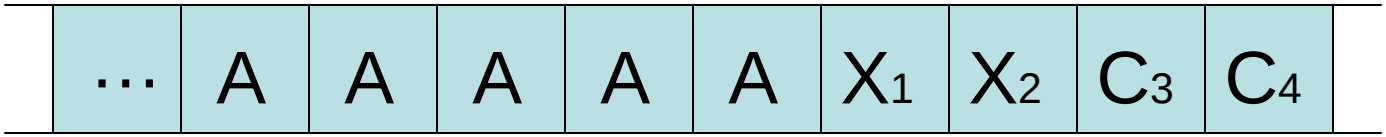
Use a type of timing attack

When brute force value is wrong, the process will exit when the function returns

When brute force value is right, the process will be allowed to return, and continue executing for some further period of time

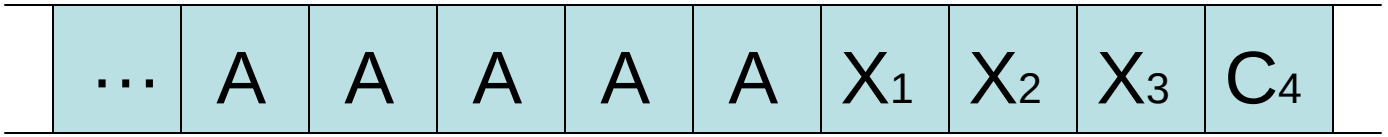
In general, incorrect guesses fail fast, correct guesses fail slow

SSP/ProPolice



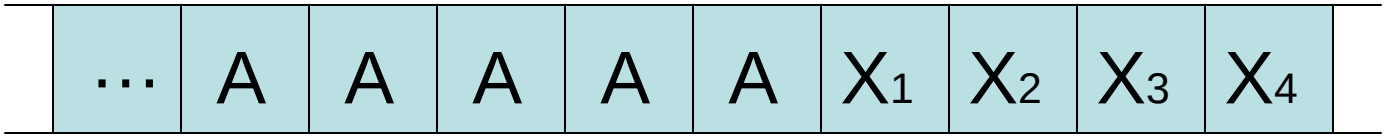
A = buffer overflow padding
X_1 = C_1, the first canary byte
X_2 = brute force byte, from 0 to 255

SSP/ProPolice



A = buffer overflow padding
$X_1 = C_1$, the first canary byte
$X_2 = C_2$, the second canary byte
X_3 = brute force byte, from 0 to 255

SSP/ProPolice



A = buffer overflow padding
$X_1 = C_1$, the first canary byte
$X_2 = C_2$, the second canary byte
$X_3 = C_3$, the third canary byte
X_4 = brute force byte, from 0 to 255

SSP/ProPolice

At the end of this process, each individual byte of the canary is known

Allows the exploit to overwrite the saved return address while still having a valid canary

Each byte takes at most 256 attempts, giving a maximum 1024 attempts for whole canary

This equates to an average 128 attempts per byte, or 512 attempts for whole canary

SSP/ProPolice

It would be better is if there was no canary at all

RJohnson and Silberman found that SSP does not guard character buffers of size less than 8

Also doesn’t guard against arrays of type other than char, such as integer and pointer arrays

Case and point: Mark Dowd’s Apache mod_rewrite bug was not prevented by SSP

SSP/ProPolice

Also very little SSP can do to guard buffers inside of a structure

Can’t rearrange structure fields, so may be able to exploit adjacent pointers etc.

But it does rearrange the structure itself (when containing a character buffer)

So in some cases it is possible to overflow the entire structure when a buffer is rearranged to lower address than structure

SSP/ProPolice

In original stack canary implementations, argument pointers could be overflowed

SSP copies the arguments to below the buffers

But what if the argument points to somewhere above the buffers?

Used when argument is for both input and output (common with structures and length variables)

Potentially exploitable in various application specific ways, by controlling input/output values at the place pointed to by the argument

ASLR

Address Space Layout Randomization (ASLR) used to obfuscate address space by adding entropy to base address of page mappings

Designed to make stable exploit development difficult: helps prevent arbitrary stack/heap overwrites, ret-to-libc etc.

OpenBSD randomizes much of its address space: library text and data segments, signal trampoline, mmap, stack

ASLR

However, parts of the address space are not randomized

Specifically, application executable's text, data and BSS segments are all mapped to static locations

The text segment includes all application code, but also Procedure Linkage Table (PLT)

The data/BSS segments contain global variables

One binary executed twice

```
0be82000-0be8a000 r-xp /usr/libexec/ld.so
0eb28000-0eb29000 r-xs
0fe66000-0fee8000 r-xp /usr/lib/libc.so.39.2
1c000000-1c001000 r-xp /root/aslr
2be82000-2be83000 r--p /usr/libexec/ld.so
2be83000-2be84000 rw-p
2be84000-2be85000 r--p
2be85000-2be87000 rw-p
2fe66000-2fe73000 r--p /usr/lib/libc.so.39.2
2fe73000-2fe76000 rw-p /usr/lib/libc.so.39.2
2fe76000-2fe78000 r--p /usr/lib/libc.so.39.2
2fe78000-2fe79000 rw-p /usr/lib/libc.so.39.2
2fe79000-2fe97000 rw-p
3c000000-3c001000 r--p /root/aslr
3c001000-3c002000 rw-p
3c002000-3c003000 r--p
3c003000-3c004000 rw-p
84981000-84982000 rw-p
854c9000-854ca000 r--p /var/run/ld.so.hints
86d90000-86d91000 r--p
cd800000-cf000000 ---p
cf000000-cf7e0000 rw-p
cf7e0000-cf7f0000 rw-p
cf7f0000-cf800000 rw-p
```

```
05d2c000-05d2d000 r-xs
09bff000-09c07000 r-xp /usr/libexec/ld.so
0eb46000-0ebc8000 r-xp /usr/lib/libc.so.39.2
1c000000-1c001000 r-xp /root/aslr
29bff000-29c00000 r--p /usr/libexec/ld.so
29c00000-29c01000 rw-p
29c01000-29c02000 r--p
29c02000-29c04000 rw-p
2eb46000-2eb53000 r--p /usr/lib/libc.so.39.2
2eb53000-2eb56000 rw-p /usr/lib/libc.so.39.2
2eb56000-2eb58000 r--p /usr/lib/libc.so.39.2
2eb58000-2eb59000 rw-p /usr/lib/libc.so.39.2
2eb59000-2eb77000 rw-p
3c000000-3c001000 r--p /root/aslr
3c001000-3c002000 rw-p
3c002000-3c003000 r--p
3c003000-3c004000 rw-p
7d99c000-7d99d000 r--p /var/run/ld.so.hints
7dbea000-7dbeb000 r--p
7e78a000-7e78b000 rw-p
cd800000-cf000000 ---p
cf000000-cf7d0000 rw-p
cf7d0000-cf7e0000 rw-p
cf7e0000-cf800000 rw-p
```

ASLR

Note that the Global Offset Table (GOT) is read only

But if an attacker can get arbitrary overwrite (via pointer or malloc chunk overwrite) then the data/BSS segments can still be used

Most applications have at least one interesting global

Similarly, arbitrary code execution may be leveraged through ret-to-text or ret-to-plt

ASLR

Other techniques can be used to find randomized mappings (i.e. libc, heap)

The most obvious being brute force

OpenBSD does most of its randomization in uvm_map_hint( )

Effectively gives 16 bits of entropy to both code and data pages (i386)

The stack is randomized elsewhere, using a “stack gap” (pages are static, stack top is not)

ASLR

It is not unreasonable to suggest brute forcing 16 bits of entropy

Gives an average of 32768 attempts (65536 max)

Locally, brute force by fixing a static address and waiting for the page to get randomized there

Forking daemon, keep trying different pages until the right one is hit…

However, not feasible if each attempt requires too much traffic (i.e. 512mb upload from 16kb per attempt)

ASLR

Introducing “data access brute forcing”

Some times a randomized mapping can be brute forced remotely in ~500 attempts, a decrease by a factor of about 100

Applicable against forking daemons (or potentially threaded apps depending on how they fail)

Still conceptual though, only really thought about this a few days ago

ASLR

The idea is similar to byte for byte brute forcing for SSP, relies on a timing attack

Traditional brute force against ASLR has worked by “code accesses” (i.e. ret, indirect call)

But mappings can also be discovered by read/write operations (read is easier)

Assume you control a pointer, and that the pointer is about to be used for a read operation

Also assume you want to find the libc text segment base

ASLR

If the pointer reads an invalid page, the process will usually fail immediately

However, a valid page will result in a successful (although nonsensical) read operation

The data will be passed back to the application, and execution continues

So successful read operations fail slower than non-successful reads

ASLR

Code access brute forcing increases page by page each attempt, using static offset in to page (i.e. offset to sleep( ))

But a read access can use increments of target's text segment size

It doesn't matter what gets read (although some successful reads may fail slower than others)

Once one mapping succeeds, test all other adjacent pages (forward and back)

This gives the start of the mapping, and the size of the mapping

ASLR

If this is possible, the numbers on OpenBSD look good:

```
0fe66000-0fee8000 r-xp /usr/lib/libc.so.39.2
```

(0xfee8000 – 0xfe66000) / 4096 = 0x82 (130)

(65536 / 130) + 130 = 635

So it takes at most 635 attempts to find libc (this figure would go up slightly if there was more than one mapping in the tested range)

ASLR

On OpenBSD it’s also possible to use this technique with a write operation

Abuse the fact that the text/data segments are mirrored, with a static difference of 512mb

Find the libc data/bss segments and start subtracting

Not as efficient as using read because data/bss segments are smaller, so need smaller increment

ASLR

Information leaks can help find an SSP canary, but can also help map randomized address space

Depends on application specific heuristics (need to leak known lib function addresses, variable pointers etc)

Also, partial overwrites can be extremely useful as the low 12 bits of any randomized address is not randomized

Allows ret-to-near, tricks with pointers, and is helpful with some malloc stuff

Malloc

OpenBSD has designed a custom memory allocator, designed for security

No in-band chunk headers

Completely mmap( ) based, which means each allocated page is randomized

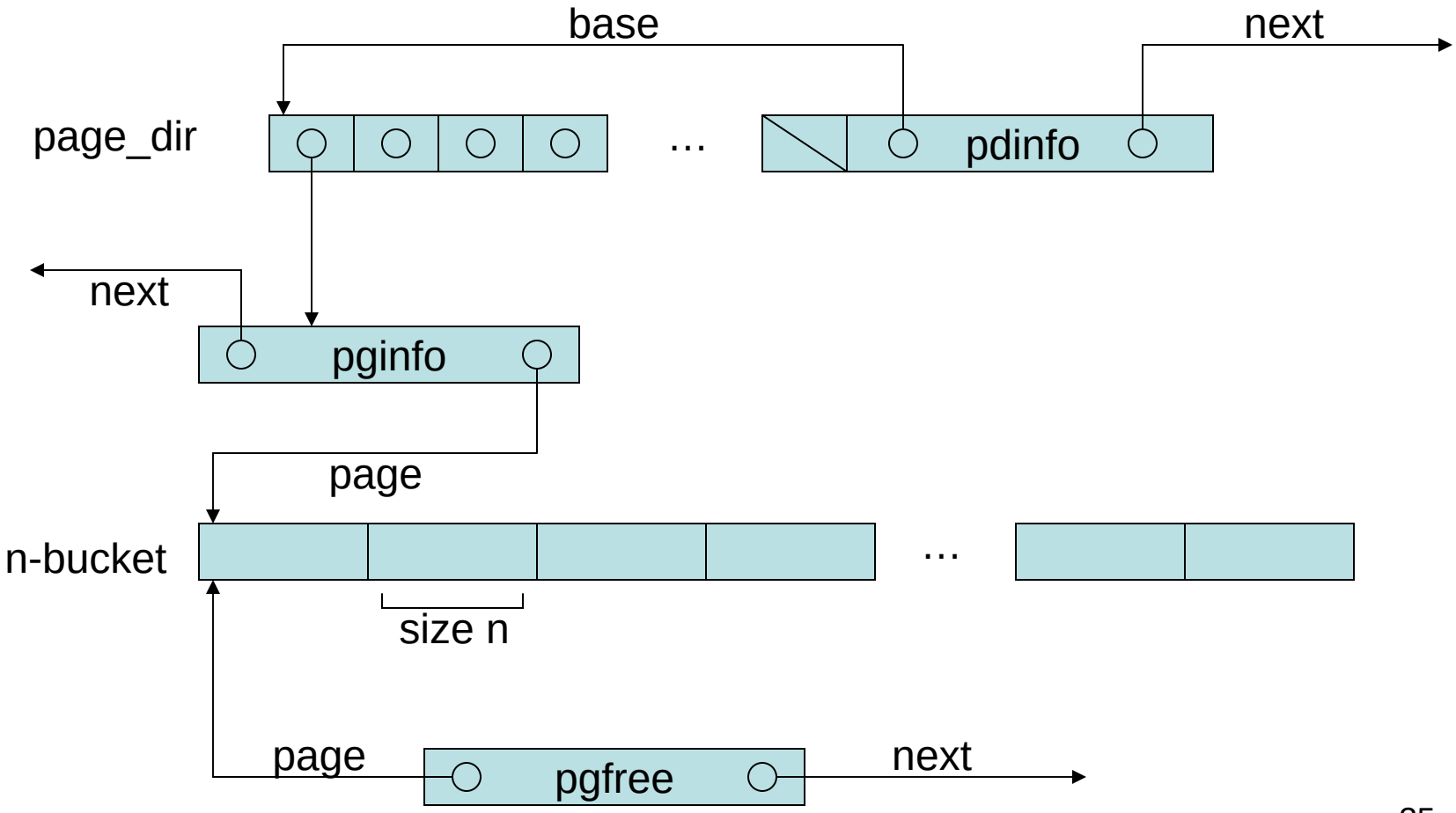
Optional support for various other improvements such as guard pages, pointer protection, junk fills

Malloc

There are 5 important parts to OpenBSD malloc:

- The page directory structure, pdinfo
- The page directories
- The chunk page structure, pginfo
- The chunk pages
- The free page structure, pgfree

OpenBSD malloc (roughly)



Malloc

Some generic techniques can be used to leverage overflows involving malloc

Most obvious is to simply overflow application specific data in an adjacent chunk

This is likely to be the most common technique, already used in some exploits (linux)

An attacker needs to position overflowed chunk to be before “target” structure

Mostly looking to overwrite function pointers, but each application is different

Malloc

There is also a generic technique involving arbitrary free( )'s

Based on the "Spirit" attack described in "Malloc Maleficarum"

Instead of leveraging the malloc implementation, leverage the application itself

Assume you control the value of a pointer as it gets passed to free( )

Malloc

Point the overflowed pointer to an existing chunk that contains an interesting structure and free it

By default, OpenBSD leaves freed memory intact

The application still thinks the chunk is allocated

If chunks of the same size are allocated, eventually the “in use but free” chunk will be allocated

If the attacker controls the use of this chunk, then the attacker controls the interesting structure

Malloc

Note that all chunks in one page are equally sized

So when partially overflowing a malloc pointer of a known size, every multiple of that size is guaranteed to be a chunk

Knowing this could often simplify the application of this technique

Credit goes to Mark Dowd for noticing this

This is complicated by the ‘J’ and ‘F’ options in OpenBSD malloc’s option file, but neither are enabled by default

Malloc

Real aim of “Exploiting Malloc” is to control heap structures which in turn allows you to control arbitrary memory

On OpenBSD this should be theoretically impossible, since there are no in-band heap structures

But this isn’t strictly true, since the allocator often uses malloc itself to allocate room for those structures

Also, there are no guard pages by default…

Malloc

Consider the pginfo structure, used to describe a page of equally sized chunk fragments

For smaller buckets (< 32 bytes), pginfo is located at the start of the page of chunks

Which is fine if you ignore the possibility of a buffer underflow

But for buckets of any other size, pginfo is allocated using imalloc…

malloc_make_chunks

```
struct pginfo   *bp
...
pp = malloc_pages((size_t) malloc_pagesize);
...
if (bits != 0 && (1UL << (bits)) <= l + l) {
        bp = (struct pginfo *) pp;
} else {
        bp = (struct pginfo *) imalloc(l);
        if (bp == NULL) {
                ifree(pp);
                return (0);
        }
}
```

Malloc

So whenever a new bucket page is created, its pginfo structure is placed into the bucket for chunks between 17 and 32 bytes in length

An attacker who can overflow a similar sized chunk that was allocated prior to new bucket page's creation can control the pginfo structure

Can do this by forcing the application to allocate a lot of similar sized chunks after overflow chunk has been allocated

Malloc

The next step is to turn the control of a pginfo structure into the control of arbitrary memory

You can do this by controlling the value returned from a call to malloc, from there it's easy

This can be achieved by supplying an arbitrary value for the "page" element:

```
struct pginfo {
        struct pginfo   *next;  /* next on the free list */
        void            *page;  /* Pointer to the page */
        u_short         size;   /* size of this page's chunks */
        u_short         shift;  /* How far to shift for this size chunks */
        u_short         free;   /* How many free chunks */
        u_short         total;  /* How many chunk */
        u_long          bits[1];/* Which chunks are free */
};
```

Malloc

The malloc_bytes( ) function uses the page element as a base after calculating an offset to the next free chunk

```
return ((u_char *) bp->page + k);
```

This can be triggered by causing an allocation of a chunk with appropriate size

Depending on which pginfo is controlled, multiple allocations may be required

The next pointer is not dereferenced here and can be safely mangled

Malloc

Also possible with a round about way through free then malloc

Overwrite only the next pointer, or overwrite entire pginfo but with null bytes for a sensible shift

Cause a free on all chunks in the page, or fake the “free” and “total” values in the pginfo

This triggers page free code, with sets the page_dir entry for this bucket to the next value from pginfo

Malloc

Point the overflown pginfo next value to a fake pginfo structure.

Then trigger a malloc for that bucket size

Bits field must be non-zero

But otherwise apply the same technique as for a straight malloc

This should only be used when necessary, the straight malloc way is much better

Malloc

The pgfree structure is used to describe a series of adjacent free pages

Similarly to pginfo, pgfree is allocated internally using imalloc

```
static struct pgfree *px;
...
if (px == NULL && (px = malloc_bytes(sizeof *px)) == NULL)
        goto not_return;
```

Allocated whenever a page becomes free

Which means an overflow in the 32 byte bucket can also smash pgfree structures

Malloc

Exploiting pgfree is more complicated than pginfo, goal is still to return arbitrary value from malloc

```
struct pgfree {
        struct pgfree   *next;  /* next run of free pages */
        struct pgfree   *prev;  /* prev run of free pages */
        void            *page;  /* pointer to free pages */
        void            *pdir;  /* pointer to the base page's dir */
        size_t          size;   /* number of bytes free */
};
```

When a pgfree structure is in use, it is stored in the free_list linked list

The malloc_pages( ) function traverses this list until it finds a page that matches the request

Malloc

If the attacker supplies an arbitrary value for the “page” element of the pgfree structure, this will be returned when the cached page is reused

The only complication is that the next and prev pointers are dereferenced for a write operation

So the first to dwords in the structure must point to valid writable memory

There is actually an unlink condition here, but non-executable pages usually makes this redundant, as the “retloc” page must be both writable and executable… W^X!

Malloc

Something important to note is that there are no guard pages by default

Normally this is not an issue, since page mappings are randomized and will mostly not be adjacent

But they are only randomized with a set amount of entropy

So leaking 128mb (average) will result in a whole lot of adjacent pages, which can potentially enable overflow's into page directory pages

Then forge pginfo structure, arbitrary malloc

Other

Kernel space bugs have become popular over the last few years

OpenBSD kernel is just as soft a target as most other operating systems

Also a mention should be given to the W^X LDT bug fixed earlier this year

Future exploitation research should also consider chroot jails, non-executable pages

There was just not enough time

Q

Questions?