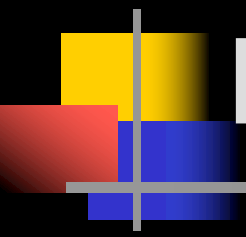
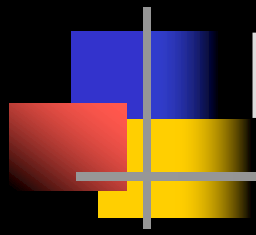


Subverting Windows 2003 SP1 Kernel Integrity Protection

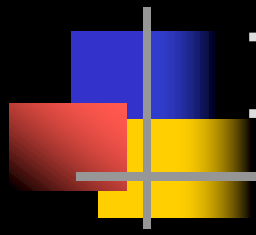
RECON 2006 - Alex Ionescu

alexi@tinykrnl.org



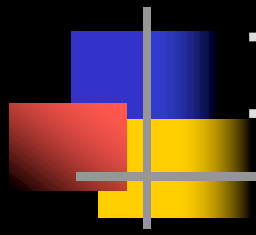
Roadmap

- Introduction
- NT Basics
  - Design
  - Objects
  - Security
- Deprecated Methods & New Methods
  - Typical (usually legitimate)
  - Atypical/specialized (usually illegitimate – rootkits)
- Subverting the protection
  - Getting access to Ring 0 memory
  - What to do with it – “payload”
- Protecting against the exploit



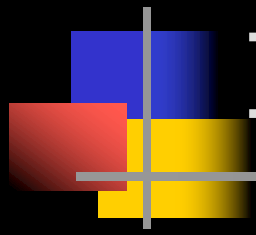
Introduction

- Simply put, Ring 0 access from Ring 3.
- Previous methods don’t work anymore.
- So => uses a newly discovered bug in the Windows NT kernel, but actual bug has been there since NT 4 or earlier.
- **Requires administrator access, not a security flaw.**



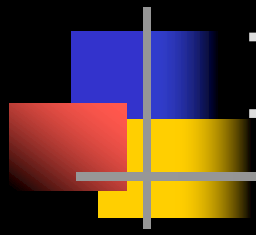
Introduction

- Goals:
  - Notifying attendees about loss of functionality in Windows 2003 SP1
    - Developers: Use newly created APIs and debugger services.
    - Security researchers: Ability to know which rootkits/malware will stop working.
  - Presenting the new exploit
    - Developers: Specialized needs might not be fulfilled by the new APIs.
    - Security researchers: Watch out for malware using this method and block it.



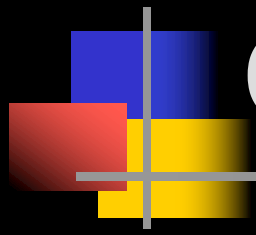
Introduction

- Usage:
  - Accessing physical memory for direct hardware access, firmware updates, BIOS/Video RAM analysis.
  - Read/write system variables (kernel) from user-mode: provide system information applications.
  - Running code at ring 0 access.



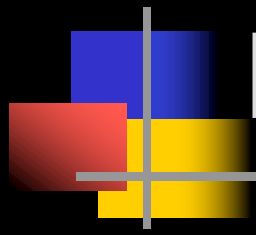
Introduction

- "But you can load a driver!"
  - User-mode malware that's actually doing kernel-mode operations is much more insidious and hard to defend against: no actual driver is in the system.
  - Loading a driver is usually detected and blocked by most IDS software.
  - Can't load a driver on 64-bit Vista: driver signing.
  - It's a lot harder to write driver code then user mode code.



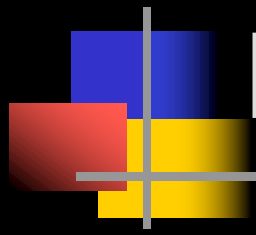
CPU Basics

- X86 supports 4 ring levels, also called CPL (Code Privilege Level) or RPL (Ring Privilege Level): Ring 0, 1, 2, 3. 3 is least privileged, 0-2 are most privileged.
- 3 blocks some operands which could crash the system or damage hardware. This is what most OSes implement as 'user mode'.
- 0-2 have mostly the same execution-level access, but memory can be segmented between them and protected. 1 and 2 are not used by any major OS: skip to Ring 0 directly.
- Ring 0: 'kernel mode' in most OSes.
- Just like the OS protects the files from guest accounts, the CPU protects the hardware from user-mode processes. Otherwise, the OS could not reliably protect against malicious assembly code which damages hardware of subverts security through hardware.
- Jump to Ring 0 is always controlled by the OS, known execution paths and heavy probing of parameters to avoid exploits. However, a privileged user can always load a driver.
- Special kind of exploits use OS-level bugs for "Ring 3 to Ring 0" privilege escalation. This means that user-mode code *directly* runs at a privileged level.
- Huge problem if it can be done from a non-privileged account, mere architectural flaw if can be done from a privileged account (since a hacker with a privileged account can already do worse things).
- However, because < Vista, rootkits would execute as Administrators under most home user's PCs, they could exploit such methods to hide from typical detection methods.



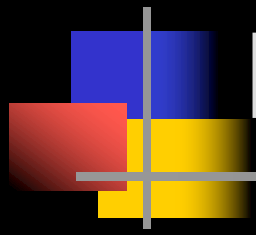
NT Basics - Design

- Hybrid kernel design (Monolithic kernel with loadable modules).
- 2 ring levels, 0 and 3, kernel and user mode, respectively.
- Multi-user, multi-session architecture.
- Object-based design, access through handles.
- Security built-in the kernel, handled by SRM (Security Reference Monitor).



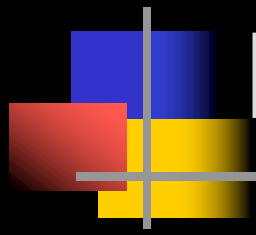
NT Basics - Objects

- Processes, threads, files, events, semaphores, mutexes are all objects managed by the Object Manager and the actual module responsible for the object's purpose.
- Objects are in kernel memory, user mode code accesses them through numerical handles, which map to the actual objects by using a per-process handle table (can have system-wide handles too).
- All objects have security information related to them, in a Security Descriptor. Object Manager talks to SRM to validate access.



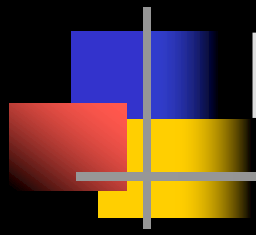
NT Basics - Objects

- Access is done by handle validation. If Process A creates an object with read access, it cannot use functions that require write access (but it can always open a new handle with write access, if the user is allowed).
- Access is also done by user validation. Process A, User Foo, can create an object and tell the kernel “User Bar can only get read access”.
- However, processes can edit permissions of other objects owned by other processes, so processes can’t fully lock each others out.



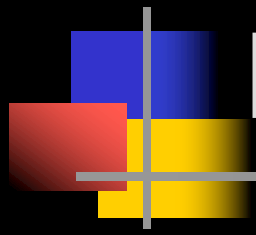
NT Basics - Security

- NT security is done with DAC: Discretionary Access Control, using ACLs: Access Control Lists. (*Vista supports M(andatory)AC too)*
- Each object has a Security Descriptor with an ACL that determines access. Each ACL has ACEs (Access Control Entries) which describe the access that a certain user can request when creating or opening a handle.
- There exist APIs for editing Security Descriptors, ACLs and ACEs, and assigning them to objects.



NT Basics - Security

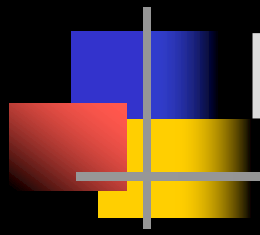
- Finally, apart from access levels and ACLs, NT has privileges and tokens.
- Tokens describe the privileges that a user has, and which ones are enabled or disabled.
- Tokens are objects too, and can be edited, as long as the user has the right access (described by the ACL).
- Some APIs only work with a certain privilege.
- More importantly, each process and thread has a token. This token describes the user.



NT Basics - Security

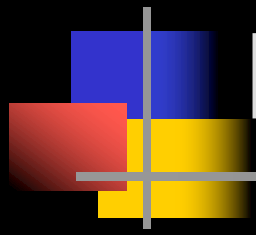
- Tokens can be duplicated, impersonated and even re-assigned. If a program launched under a guest account duplicates the token of the System process (running at highest privileges), and assigns it to itself, then this program now has full privileges too.
- It can also impersonate the token and use it for privileged APIs or access to secure objects.

Deprecated Methods and New Methods



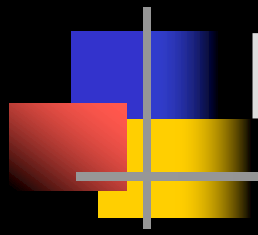
- x86 architecture allows for two typical methods of elevating ring privilege:
  - Callgates
  - Interrupts
- Usually interrupts jump to a specific location kernel-mode, but they can be created with a pointer inside a user-mode program.
- However, callgates and interrupts can only be created from kernel-mode, because they’re located in Ring 0 memory.

Deprecated Methods and New Methods



- Backdoor exists in Windows NT, similar to Linux.
- \Device\PhysicalMemory, similar to /dev/mem, allows user-mode to access physical memory.
- Problem: structures needed to add interrupts/callgates are in virtual memory => need a way to map them (paging / page tables).
- Mapping usually done by CPU using a table. Doing it manually requires reading the table, but the table is in kernel mode memory too!

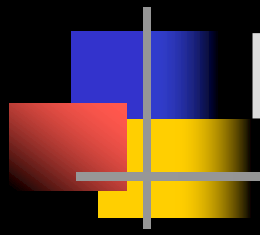
Deprecated Methods and New Methods



- So how can we do the mapping? One way would be a brute force scanning approach with a known value at a known virtual address. However, the mapping relation can change even between pages (4096 bytes).
- Solution: use a side-effect of the NT Memory Manager/Model. For *most* kernel pages, the following formula can be used:

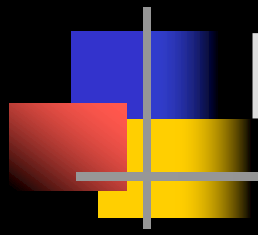
```
P = V & 0x1FFFFFFF;
```

Deprecated Methods and New Methods



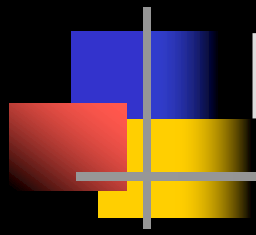
- Generic methods seen in malware and low-level software:
  - Use the \Device\PhysicalMemory object to edit the structures for interrupts and callgates (IDT and GDT) and add a callgate or interrupt. Code now runs in Ring 0.
  - Use it to edit kernel structures, such as the process list to hide processes.
  - Use it to edit the code inside Kernel APIs that return files or processes in order to hide certain ones.
  - ..etc…
  - Legitimate use: use the object to read the BIOS and/or Video ROM for system identification, or read kernel variables to provide deep, low-level system information without using a driver.

Deprecated Methods and New Methods



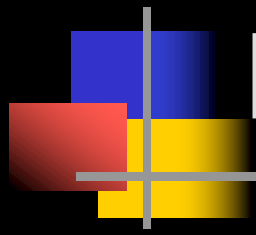
- Microsoft did not originally respond to \Device\PhysicalMemory. Some IDS software blocked access to the object through hooking.
  - Valid reason for ignoring the issue: object only accessible as an administrator. If a malicious user is already an administrator, then he can already load a driver, format the disk, etc.
- Ironically, they added a *new* vector: ZwSystemDebugControl.

Deprecated Methods and New Methods



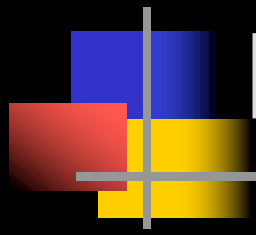
- The API allows direct access to kernel memory through the virtual address (no need for mapping and using physical memory).
- Allows sending interrupts, taking to hardware, even reading and writing MSRs (highly critical CPU registers).
- Was developed for a new feature in XP: Live Kernel Debugging. The user-mode debugger needed the API for most of its commands.
- Within a few months, was discovered and came into use by malware and low-level software.

Deprecated Methods and New Methods



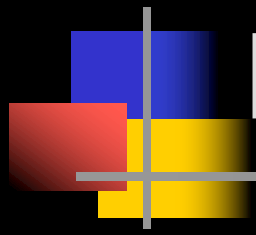
- Again, this needs administrator access. The only extra requirement was a special privilege (recall the security intro), which can be acquired with a single call.
- Was not considered a threat by Microsoft, and its usage is relatively low in the wild.

Deprecated Methods and New Methods

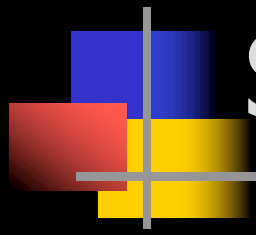


- Windows 2003 shipped with these features intact. However, by the time SP1 was being developed, XP SP2 was in development, with a heavy importance on security.
- Microsoft removed *both* threats in Service Pack 1.
- Was not a silent removal. Microsoft statement: "This change was made to help prevent security exploits that might leverage the functionality of the \Device\PhysicalMemory object from user-mode. "
- Revealed an interesting change in mentality: due to the prevalence of rootkits and most Windows users running in the administrators' context, the "administrator can do anything" concept became "any Windows application can do anything".
- Probably will revert back to the "root = God' idea once Vista ships: users won't be administrators anymore.

Deprecated Methods and New Methods

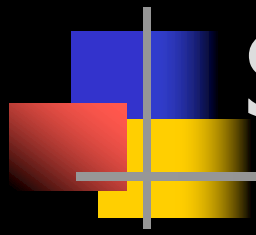


- Removing access to \Device\PhysicalMemory caused compatibility breaks with legitimate software that needed access to SMBios, ACPI, BIOS/Video ROMs, etc.
- SP1 added two new APIs to handle them: EnumSystemFirmwareTables and GetSystemFirmwareTable.
- However, writing is still disabled, and access to exotic (non SMBios/ACPI/BIOS memory) now requires a kernel mode driver.
- Is this the end of user-mode rootkits with Ring 0 access?



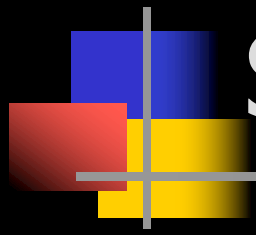
Subverting the protection

- Keystone: VDM, Virtual Device Machine, the built-in virtualizer in NT responsible for DOS support (based on SoftPC).
- VDM processes need direct Video RAM access for write/read. This memory is physical, and needs to be mapped.
- Kernel maps the Video RAM segment in a VDM process (and other BIOS ROM data).



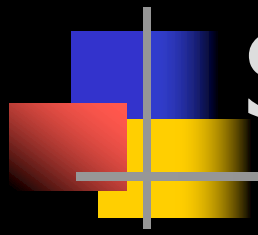
Subverting the protection

- Kernel needs to know “which addresses should I map?”
- Data located in a registry key: HKLM\HARDWARE\DESCRIPTION\Configuration Data
- Stored in CM_ROM_BLOCK structure, documented and explained in DDK. Very simple pair of Base and Size data.
- Seems simple: edit Base/Size to what we need to access.



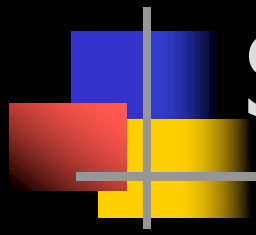
Subverting the protection

- Reality is very complex:
  - Designers had some idea of the risks: addresses below 0xC0000 are skipped. Some static kernel structures are safe: KUSER_SHARED_DATA, KPCR, Page Table Directory, IDT, GDT, more… So we can only modify actual code or structures inside the kernel.
  - Remember, this is physical memory. We need to depend on the mask shown earlier, but it's not reliable on anything else then the base kernel pages.



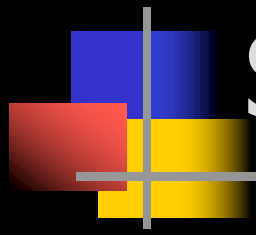
Subverting the protection

- It gets worse:
  - How on earth do we initialize as a VDM process?
    - NtVdmControl(VdmInitialize…);
  - It fails! It can only be called by a VDM process! Chicken and egg problem?
    - NtSetInformationProcess(ProcessWx86Information…);
  - It fails! It can only be called by a process with the Subsystem Privilege (recall Tokens).
    - RtlAcquirePrivilege(SeTcbPrivilege…);
  - It fails!



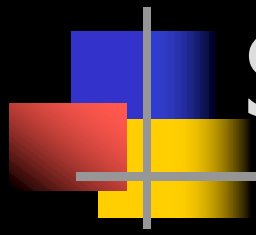
Subverting the protection

- More woes:
  - TCB (Subsystem) Privilege is only present for the Local System account, not even administrators. (But… Recall that tokens can be duplicated for impersonation!)
    - NtOpenToken(TOKEN_DUPLICATE_ACCESS…);
  - It fails! System process token was not created with this access level for the Administrator account (recall ACLs).
    - We'll have to edit the Token…



Subverting the protection

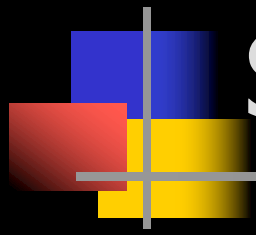
- So, our general plan of attack:
  - Open the system process' token for ACL edit access (finally, something that doesn't fail).
  - Give the administrator account access to TOKEN_DUPLICATE_ACCESS through ~20 or so complex ACL editing APIs.
  - Re-open the token with duplicate access and call NtDuplicateToken to duplicate it as an impersonation token.
  - Impersonate the current thread as a Local System Thread.
  - Acquire Set Primary Token Privilege, required to set a new process token.
  - Duplicate the token again, as a primary token.
  - Call NtSetInformationProcess to set the token as primary.
  - Acquire TCB privilege, now call NtVdmControl.
  - … you guessed it… it fails!



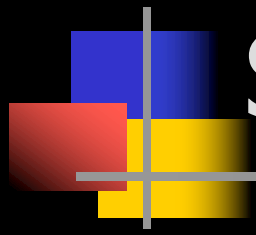
Subverting the protection

- Function probes parameters, so that kernel-mode doesn't crash while attempting to read them, but we were sending NULL data.
- Reversed NTVDM.exe to figure out the data structures needed for a "stub" initialization (ie, the minimum number of valid data).
- Called NtVdmControl with this new structure… it fails!
- VDM wants the memory at 0x0 to be already allocated so that it can copy some data inside.

Subverting the protection



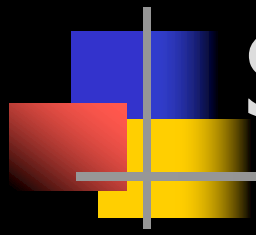
- How to allocate memory at 0x0? A NULL pointer to NtAllocateVirtualMemory has another meaning. Easy solution: use 0x1.
- Once allocated, NtVdmControl might *still* fail: the memory it tries to map might not be free.
- Therefore, we need to make sure it's free, and release it beforehand. Could be a problem if memory allocated is critical.
- Better solution: don't use any other DLLs then ntdll, and calculate the address you need beforehand, and immediately allocate it. When calling NtVdmControl, free it right before.



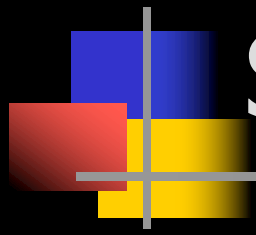
Subverting the protection

- Attempted the NtVdmControl call once more… STATUS_SUCCESS!
- 0xC0000, 0xD8000, 0xE0000 were mapped.
- Now that this works, actual flaw can be exploited.

Subverting the protection

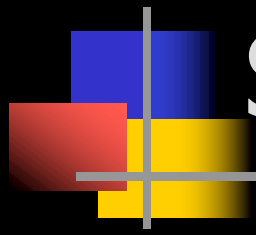


- Calculate the physical pointer for the virtual location of the code you want to modify, or data you want to read, and add a new CM_ROM_BLOCK in the registry.
- Allocate 0x0-0x10000 and allocate the pointer you'll need.
- Do Token Kung-Fu to get TCB privilege.
- Free the pointer you'll need and call NtVdmControl.
- Write your patch…
- … it fails! Actually, it doesn't, but the changes don't show up.
- How do we solve this latest problem?



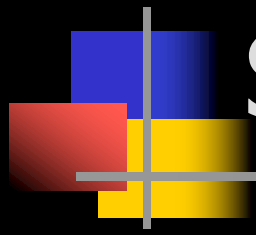
Subverting the protection

- Writes seem “lost” because the memory was allocated by the kernel, in kernel mode, with kernel flags. Therefore, user mode has no access to it, from the CPU’s perspective, so they are silently ignored.
- Yet another ace in the hole: NtWriteVirtualMemory.
- API does actual write in kernel mode, so CPU sees the access as being from Ring 0: writes happen.
- Slightly more annoying then being able to do direct memory writes (overhead of an API), but thankfully, it works!



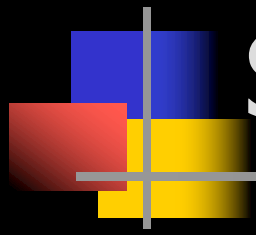
Subverting the protection

- Now that we can patch kernel code, what can we do?
- Best solution: restore lost functionality in SP1. Once \Device\PhysicalMemory access is restored, no more need to use this length hack. Even better, restore ZwSystemDebugControl.
- Part two: analyzing how \Device\PhysicalMemory became protected.



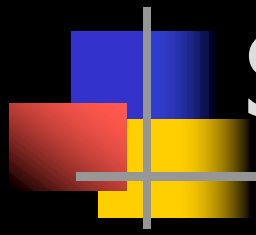
Subverting the protection

- Microsoft states that even administrators can’t open the object anymore, and that the protection is specifically for user-mode access, in general.
- Confirmed by looking at ACL of the object: matches the one in previous versions of NT.
- Therefore, either a special flag is on the object (makes sense), or a string compare is done in the kernel (ugly and problematic: could potentially use a symbolic link to bypass).



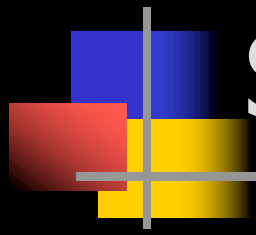
Subverting the protection

- First function analyzed: MiInitializePhysicalMemorySection. Creates the memory section with ObCreateObject, just like any other object.
- However, a strange flag is used on ObjectAttributes.Attributes: 0x10000.
- Attributes are defined in ntdef.h. Highest one has always been 0x400, so this flag definitely stands out.
- Wrote a test driver that created a new object, using this flag: object became protected as well.
- This demonstrated a new kernel check for this flag, or something similar and generic.



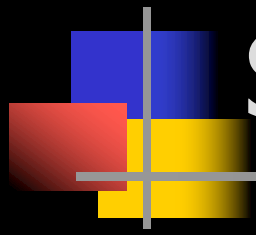
Subverting the protection

- Objects have meta-structures which are internally used by the Object Manager to handle them (for example, the reference count).
- Most common is OBJECT_HEADER. Contains sub-headers such as OBJECT_HEADER_NAME_INFORMATION.
- However, the 0x10000 value was nowhere to be seen, indicating an internal conversion to some other value stored somewhere. Typically, this is done in ObjectHeader->Attributes, but this was not the case!



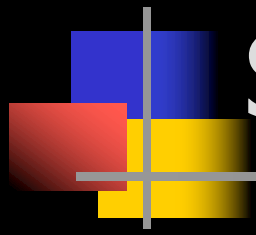
Subverting the protection

- Quickest way to find internal flag was runtime analysis: traced execution if handle open attempt while checking for EAX = STATUS_ACCESS_DENIED.
- Found where EAX was being set to this, then looked at jumps to the location and quickly identified suspect code.
- It checked for 0x40000000 in an object header structure, but only if the call came from user-mode.



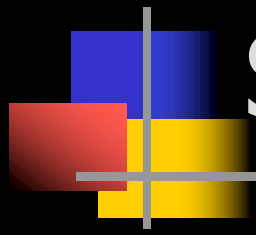
Subverting the protection

- Turns out this is OBJECT_HEADER_NAME_INFORMATION's QueryReferences member (which existed before).
- Seems to clearly be an obfuscation technique, by masking a non-related object header value with this flag.
- Experimented: removed the flag while Windows was running and tried to open the handle again: access was allowed. Likewise, masking another object with it caused it to be protected.



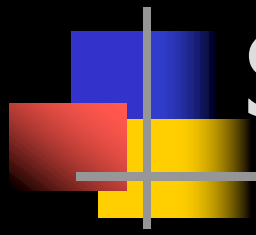
Subverting the protection

- Feature is not exploitable: Kernel only allows creating such an object from kernel mode, so user mode can’t create stealth processes or threads.
- However, driver developers can now use this flag to protect their objects from user-mode.
- Armed with the VDM bugs, two possibilities exist: get the object’s pointer and remove the flag, or edit the Object Manager and remove the check.



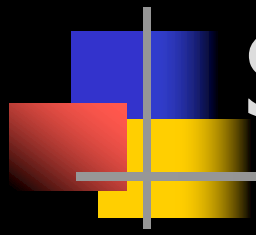
Subverting the protection

- There isn’t any way of getting the object’s pointer, and it changes at every startup.
- On the other hand, the kernel code responsible for the check can either be:
  - Hardcoded based on a table of known versions
  - Found dynamically through semi-complex regexp matching.



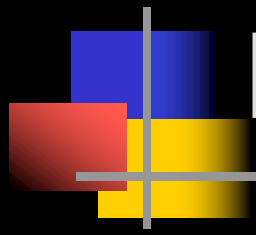
Subverting the protection

- Ultimately comes down to replacing a JNZ with a NOP or replacing it with a JMP, depending on how the target kernel was compiled (ie: which version of the kernel).
- Because the kernel can be mapped in user-mode, memory search can be done quickly and easily, and the right patch to be applied found.
- Then, use VDM exploit and use NtWriteVirtualMemory to write the NOP or JMP.
- Now we're back to < SP1 days, and \Device\PhysicalMemory is back in business.



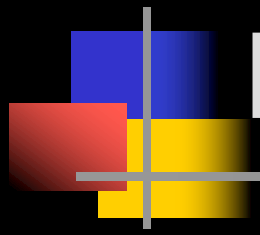
Subverting the protection

- More fun: Now use \Device\PhysicalMemory to re-activate ZwSystemDebugControl.
- Not in too many details, API was “disabled” by ripping out its code and adding a new Kernel API, KdSystemDebugControl.
- WinDBG now ships with a driver, calls the driver, and then the driver calls KdSystemDebugControl. Driver isn’t there by default, so the API is unreachable directly.
- However, a simple 6 line patch to ZwSystemDebugControl can be applied to have it call KdSystemDebugControl: functionality is fully restored.



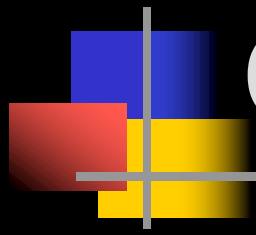
Protecting

- **Do not let non-trusted users logon as administrators (or power users or any other privileged account).**
- If you absolutely must, or are writing software for home users which are running as administrators by default, you can use the Registry Notification APIs to monitor an application editing the key. Windows only does at startup, so anyone else is most probably fishy.
- Microsoft was notified of the flaw, so a fix might be released.



Paper/Proof of Concept/Greets

- More detailed paper (over 40 pages) with actual assembly analysis, object dumping and more NT design explanations will be available within the week at: http://www.tinykrnl.org/recon2k6.pdf
- Paper contains inlined sample code which matches internal proof of concept.
- Thanks to wtbw and Jason Geffner for their help.
- And finally, thanks to all the audience!



Questions/comments

- Now (if time allows).
- After the talk.
- By email: alexi@tinykrnl.org
- **Will not respond to questions that allow exploit to be used in malware.**