Bypassing Memory Protections: The Future of Exploitation

Alexander Sotirov

alex@sotirov.net

About me

- Exploit development since 1999
- Research into reliable exploitation techniques:
  - Heap Feng Shui in JavaScript
  - Bypassing browser memory protections on Windows Vista (with Mark Dowd)
- Part of the team that created a rogue CA using an MD5 collision last year

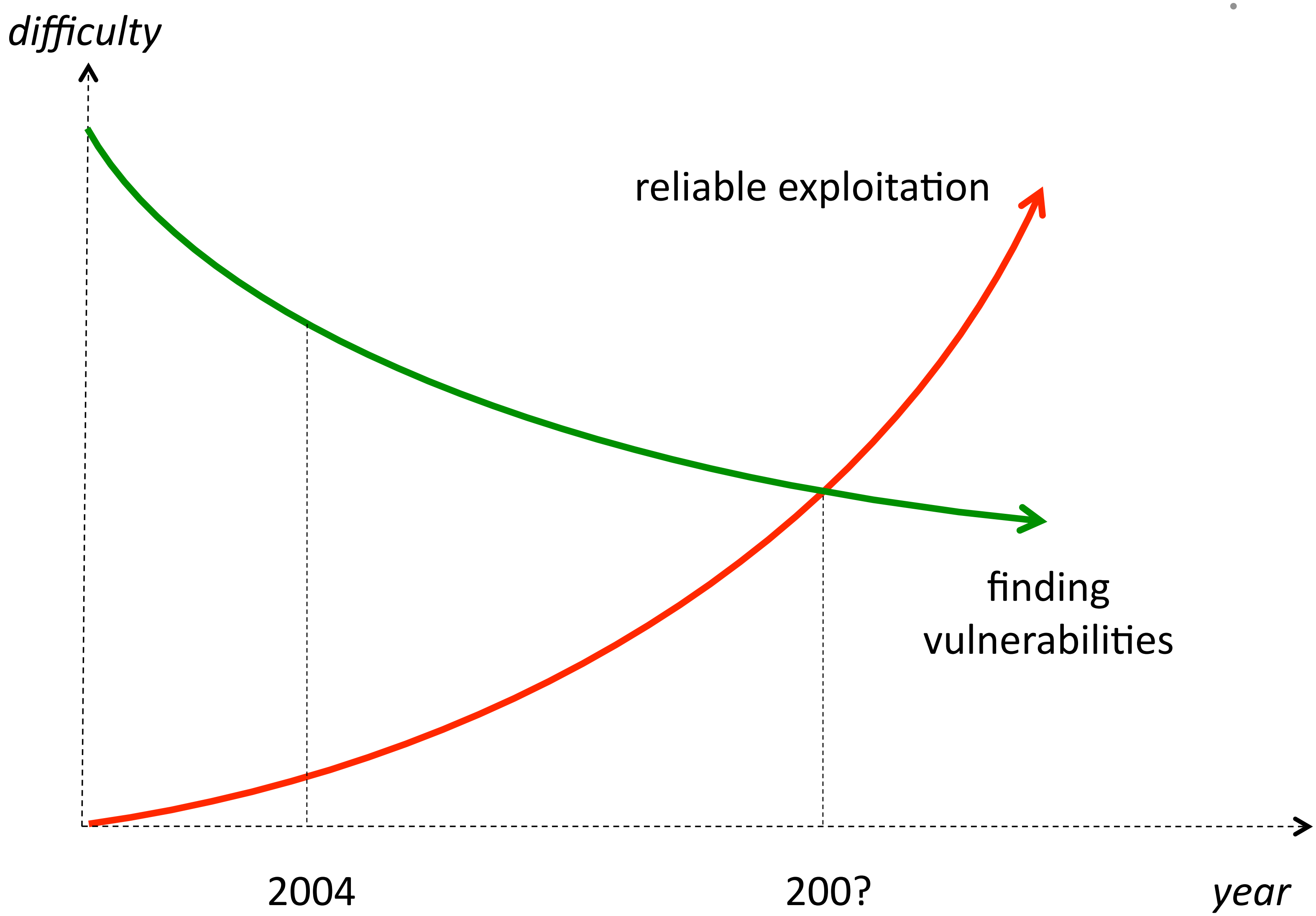
Definitions

Exploit:

a program that generates data to trigger a vulnerability and achieve **reliable** arbitrary code execution or subversion of the application logic

This talk covers only exploits for memory corruption vulnerabilities.

Exploitation is getting harder



Spending several man-months to turn a crash into an exploit is not unusual.

Overview of this talk

- Exploitation back in the summer of 2004
- The evolution of exploit mitigations
  - GS
  - DEP
  - ASLR
  - SafeSEH
- State of the art in exploitation
- The future of exploitation

Part I

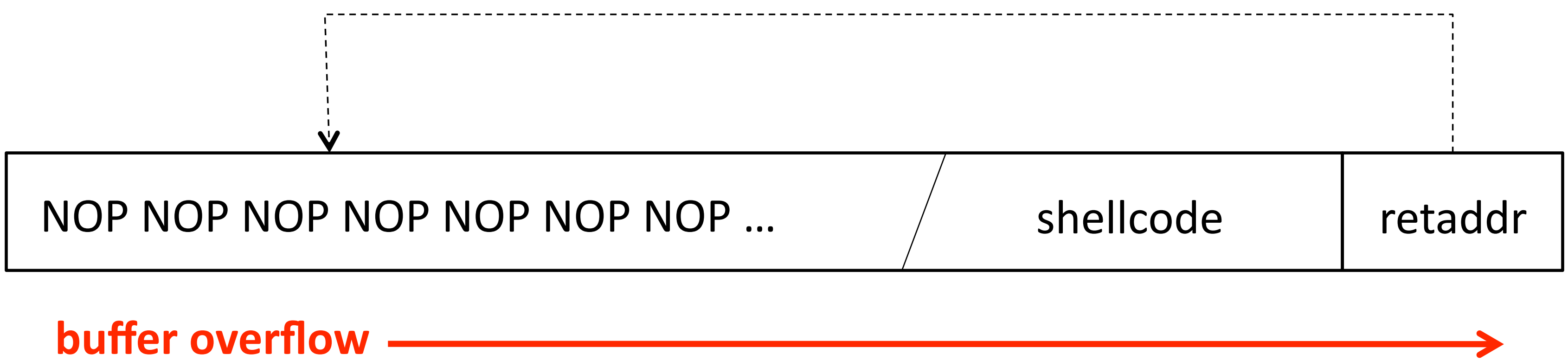
The summer of 2004

State of exploitation in 2004

- All major C vulnerability classes were already well known:
  - stack overflows
  - format string bugs
  - heap overflows
  - integer overflows, signedness issues
- Fuzzing made vulnerability discovery easy
- From the mid 1990s until 2004 we could exploit anything!

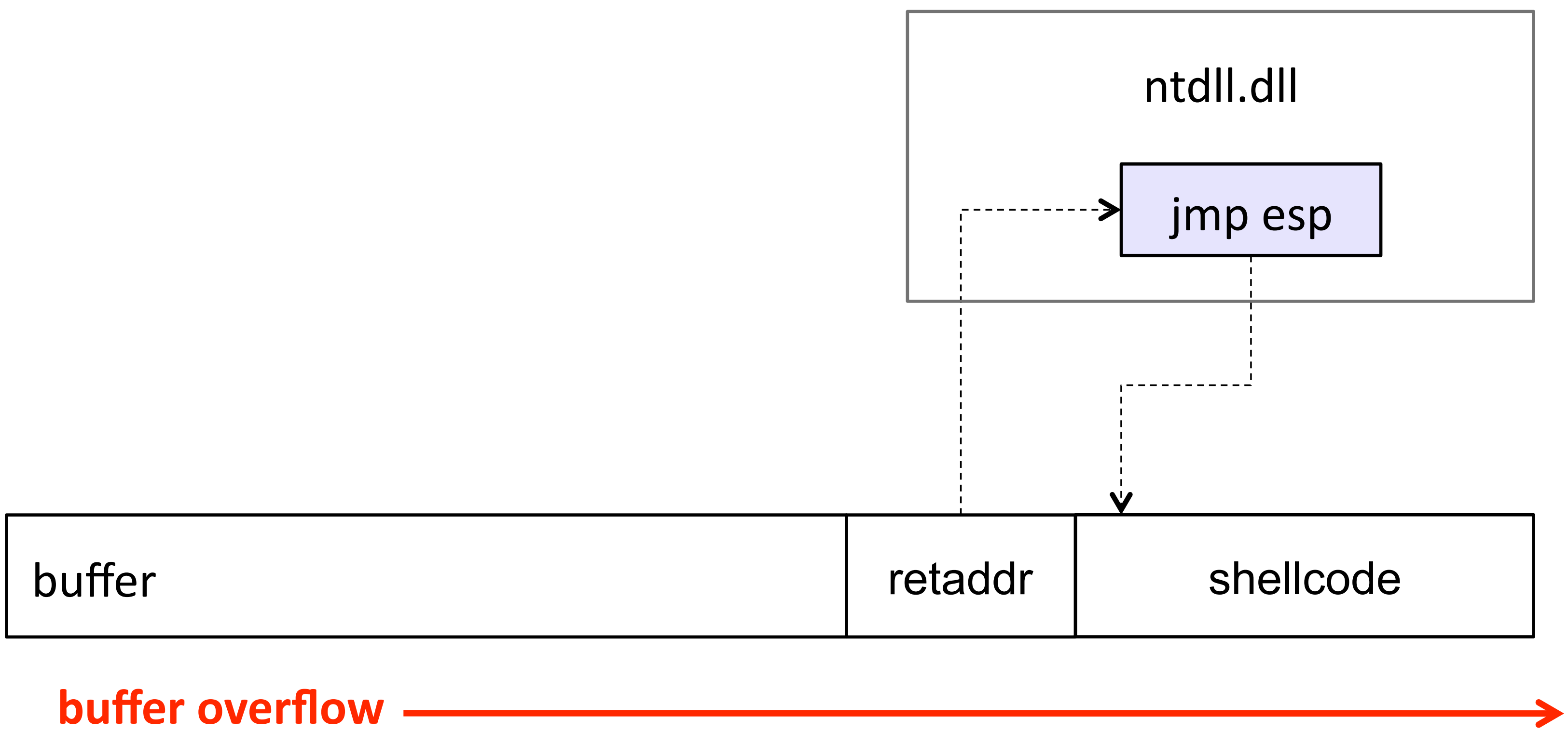
Stack overflows on Linux

Linux single-threaded application with a static stack base address:



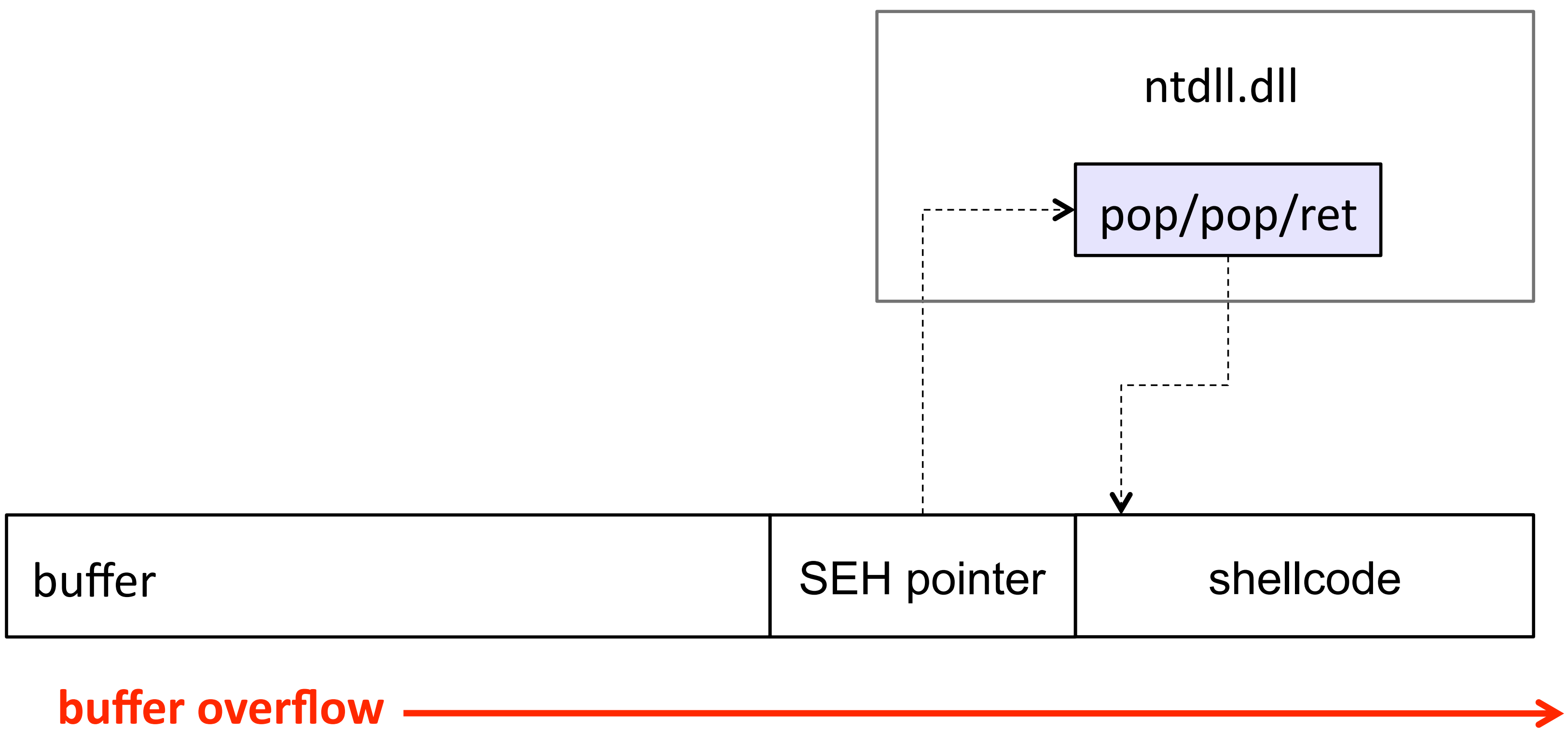
Stack overflows on Windows

Windows multi-threaded application, ntdll.dll loaded at a static base address:



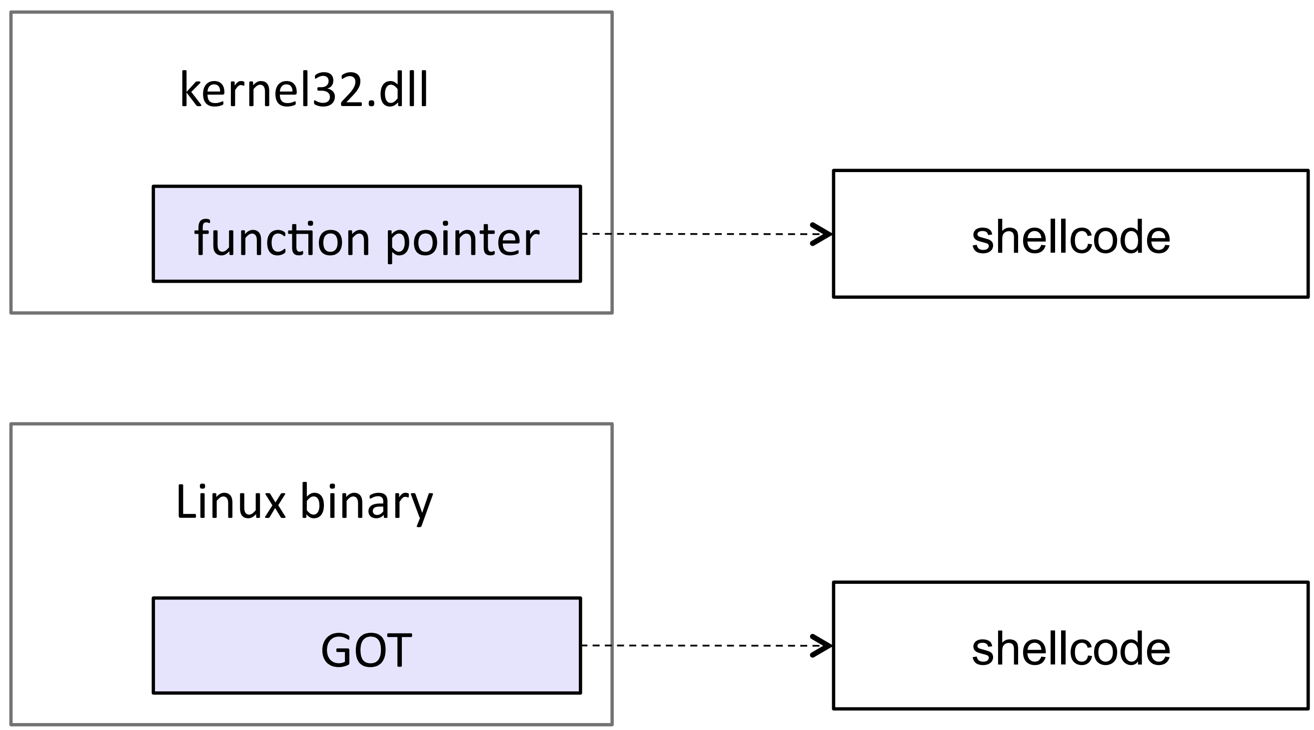
Stack overflows on Windows

Windows SEH pointer overwrite followed by access violation before the function returns:



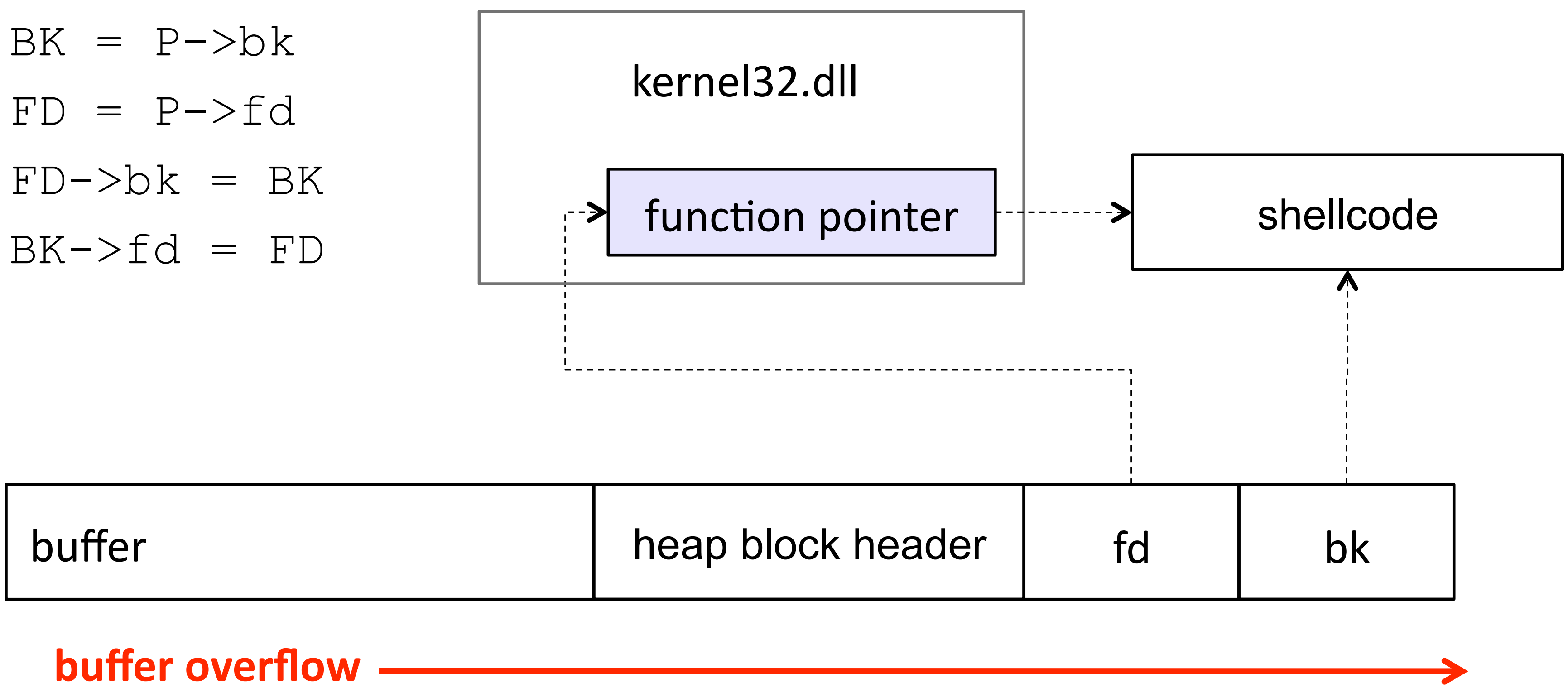
Format string bugs

%n allows us to write an arbitrary 32-bit value to an arbitrary address:



Heap unlink exploitation:

```
BK = P->bk
FD = P->fd
FD->bk = BK
BK->fd = FD
```



OS features we could rely on

- Fixed addresses of stack and executables
  - we can place shellcode on the stack or jump through a `jmp reg` trampoline in a binary
- Function pointers at well-known locations
  - great targets for arbitrary memory writes
- Heap allocator that trusts heap metadata
  - generic way to turn heap overflows into arbitrary memory writes
- Executable data on the stack and heap
  - easy to execute shellcode

The beginning of the end

- Windows XP SP2 (Aug 2004)
  - Non-executable heap and stack
  - Stack cookies
  - Safe unlinking
  - PEB randomization
- RHEL 3 Update 3 (Sept 2004)
  - Non-executable heap and stack
  - Randomization of libraries

Part II

The Evolution of Exploit Mitigations

OS evolution

	XP SP2, SP3	2003 SP1, SP2	Vista SP0	Vista SP1	2008 SP0
GS					
stack cookies	yes	yes	yes	yes	yes
variable reordering	yes	yes	yes	yes	yes
#pragma strict_gs_check	no	no	no	?	?
SafeSEH					
SEH handler validation	yes	yes	yes	yes	yes
SEH chain validation	no	no	no	yes [1]	yes
Heap protection					
safe unlinking	yes	yes	yes	yes	yes
safe lookaside lists	no	no	yes	yes	yes
heap metadata cookies	yes	yes	yes	yes	yes
heap metadata encryption	no	no	yes	yes	yes
DEP					
NX support	yes	yes	yes	yes	yes
permanent DEP	no	no	no	yes	yes
OptOut mode by default	no	yes	no	no	yes
ASLR					
PEB, TEB	yes	yes	yes	yes	yes
heap	no	no	yes	yes	yes
stack	no	no	yes	yes	yes
images	no	no	yes	yes	yes

Exploit mitigations

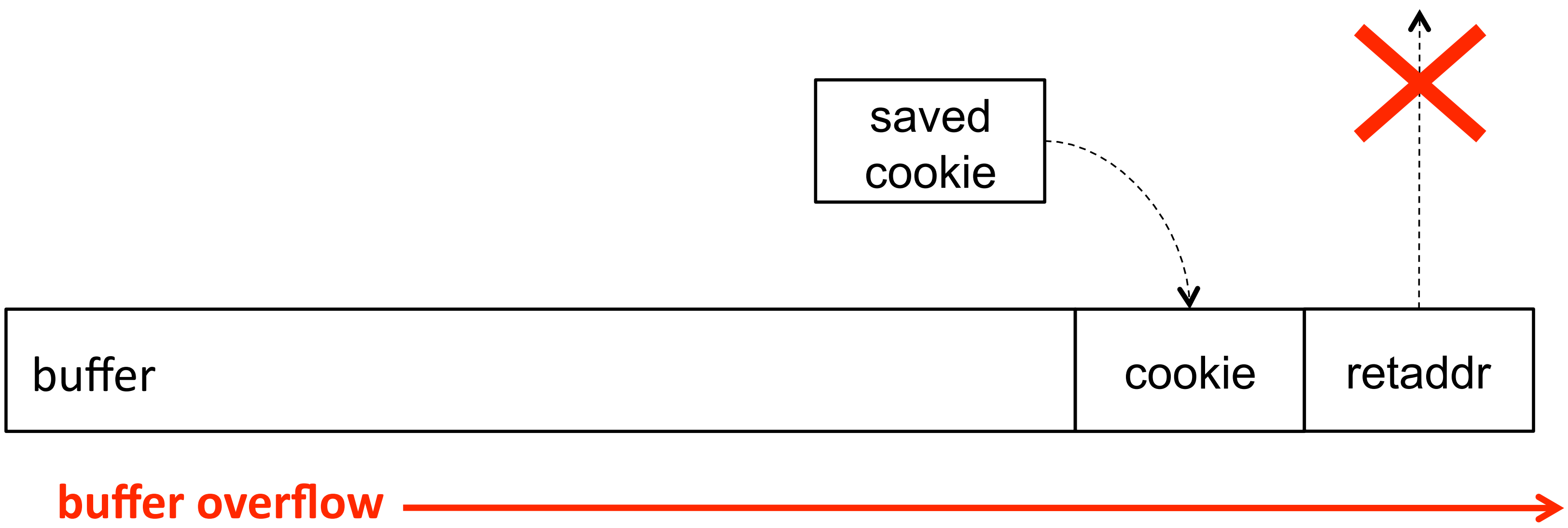
Detect memory corruption:

- GS stack cookies
- SEH chain validation (SEHOP)
- Heap corruption detection

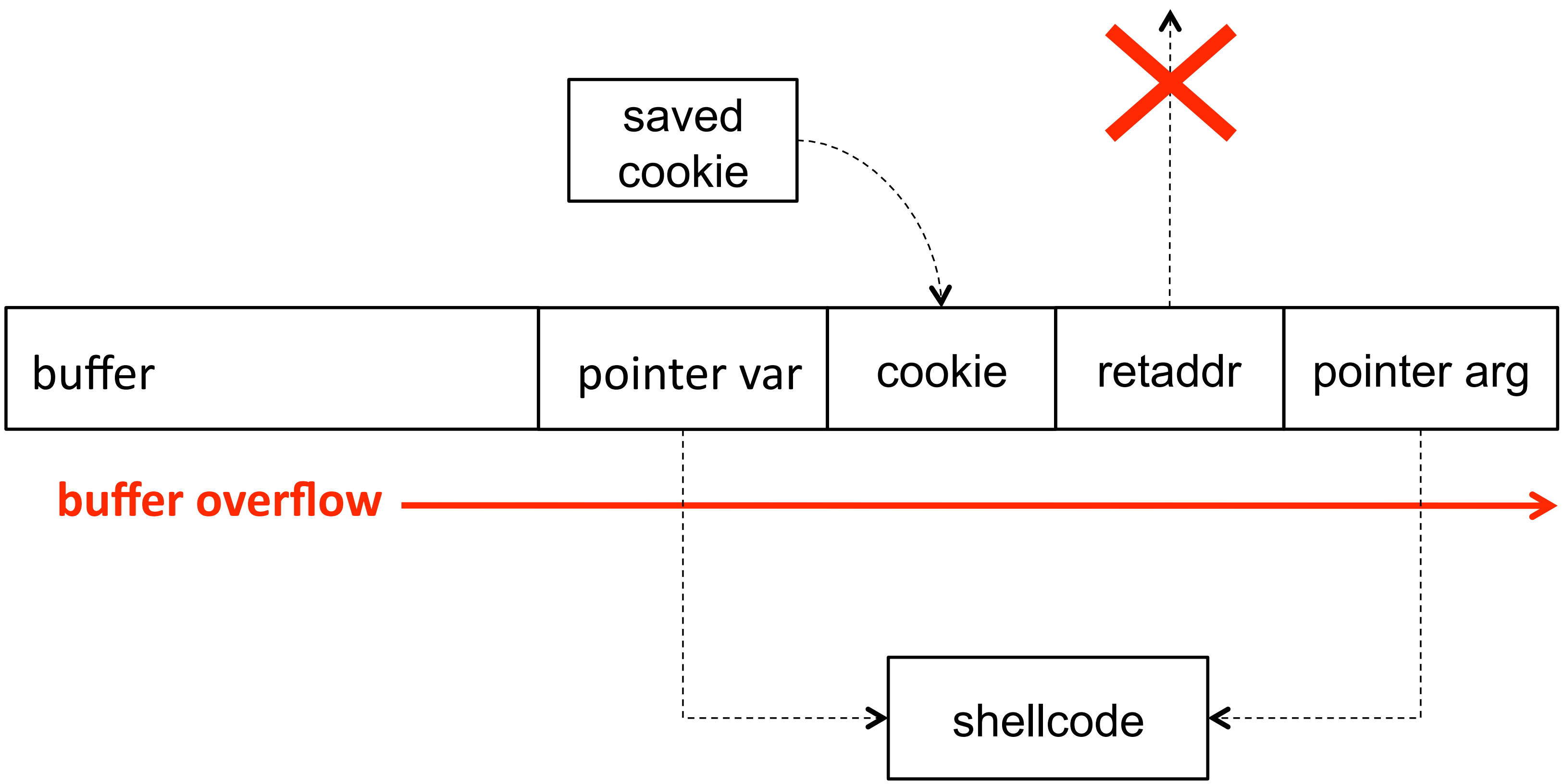
Stop common exploitation patterns:

- GS variable reordering
- SafeSEH
- DEP
- ASLR

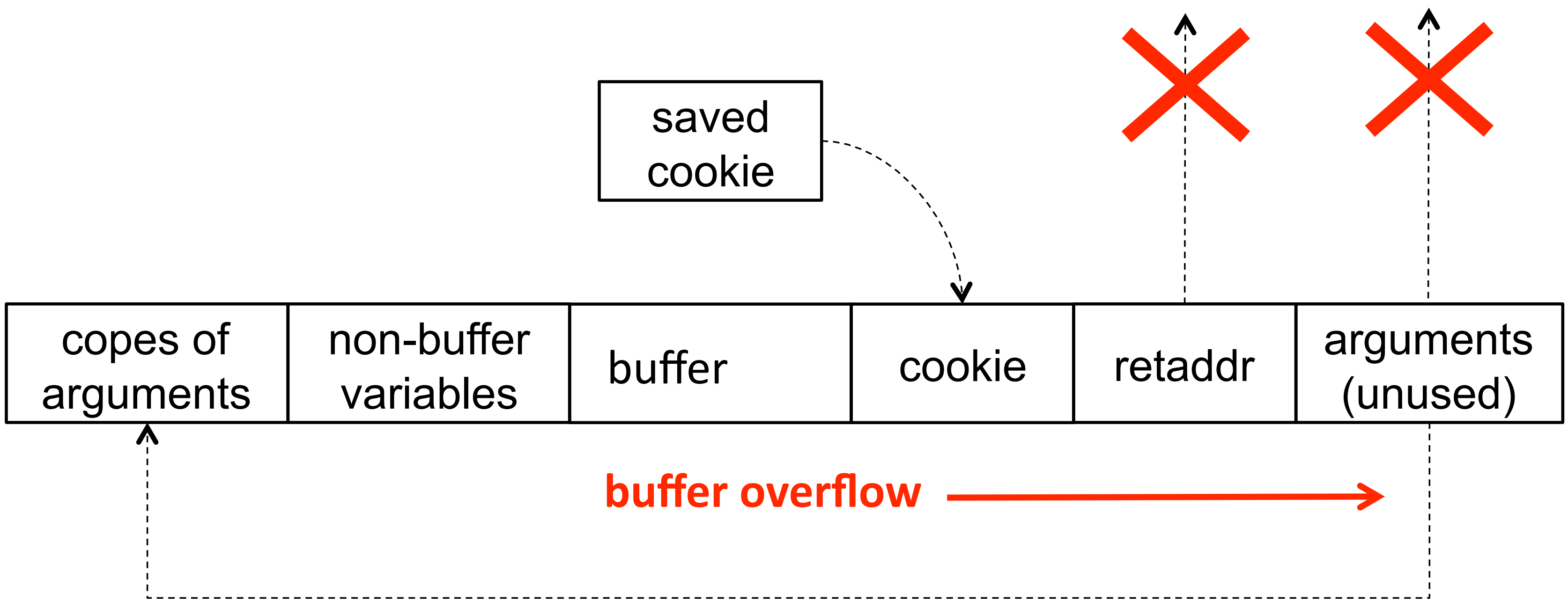
GS stack cookies



Breaking GS


saved
cookie
buffer
pointer var
cookie
retaddr
pointer arg
buffer overflow
shellcode

GS variable reordering



pointer arguments are copied
before the other variables

Breaking GS, round 2

Some function still use overwritten stack data before the cookie is checked:

```
callee saved registers
copy of pointer and string buffer arguments
local variables
string buffers                  o
gs cookie                       v
exception handler record        e
saved frame pointer             r
return address                  f
arguments                       l
                                o
stack frame of the caller       w
```

SafeSEH

- Validates that each SEH handler is found in the SafeSEH table of the DLL
- Prevents the exploitation of overwritten SEH records

Breaking SafeSEH

- Requires that all DLLs in the process are compiled with the new /SafeSEH option
- A single non-compatible DLL is enough to bypass the protection
- Control flow modification is still possible

SEH chain validation (SEHOP)

- Puts a cookie at the end of the SEH chain
- The exception dispatcher walks the chain and verifies that it ends with a cookie
- If an SEH record is overwritten, the SEH chain will break and will not end with the cookie
- No known bypass techniques

Data Execution Prevention

- Executing data allocated without the PAGE_EXECUTABLE flag now raises an access violation
- Stack and heap protected by default
- Prevents us from jumping to shellcode

Breaking DEP

- Off by default for compatibility reasons
- Compatibility problems with plugins: Internet Explorer 8 finally turned on DEP
- Sun JVM allocated its heap memory RWX, allowing us to write shellcode there
- Return oriented shellcode (ret2libc)
    - DEP without ASLR is completely useless

ASLR

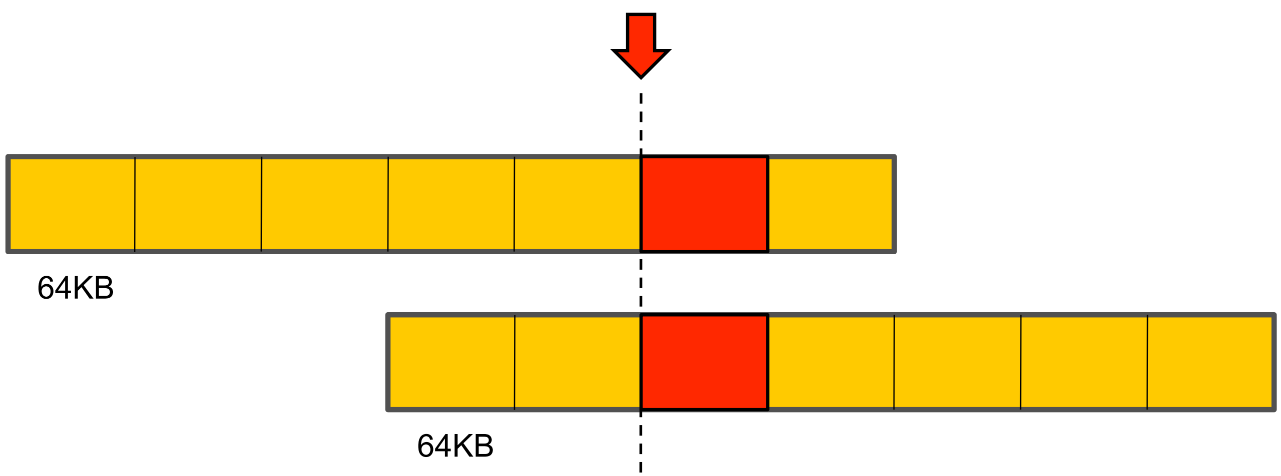
- Executables and DLLs loaded at random addresses
- Randomization of the heap and stack base addresses
- Prevents us from jumping to existing code

Breaking ASLR

- Enabled only for binaries compiled with a special flag (for compatibility reasons)
- Many browser plugins still don't have it
- Heap spraying still works
  - ASLR without DEP is completely useless

Breaking ASLR

- Heap spraying defeats ASLR
- 64KB-aligned allocations allow us to put arbitrary data at an arbitrary address
    - Allocate multiple 1MB strings, repeat a 64KB pattern



Part III

State of the art in exploitation

Windows pre-XP SP2

- Exploitation is trivial
- Multiple tools automate the process of analyzing a stack overflow crash and generating an exploit
- Nobody cares about these old systems

Windows XP SP2

- The most widely targeted system in mass exploitation for botnets and keyloggers
- Attack surface reduction has reduced the number of vulnerabilities in services, but client software is almost completely unprotected
- Reliable exploitation techniques exist for almost all types of vulnerabilities

Windows Vista

- Limited deployment, not a target for mass exploitation yet
- More attack surface reduction in services, but client software still an easy target
- ASLR and DEP are very effective in theory, but backwards compatibility limitations severely weaken them

Windows 7

- Minor exploit mitigation changes since Vista (as far as I know)
- Potential for a wide deployment
- Improved support for DEP and ASLR from Microsoft and third party vendors:
  - .NET framework 3.5 SP1
  - Internet Explorer 8
  - Adobe Reader 9
  - Flash 10
  - QuickTime 7.6

Part III

The future of exploitation

Is exploitation over?

What if all software used these protections to the fullest extent possible?

Assume a Windows 7 system with the latest versions of all common browser plugins.

Partial overwrites

- Windows binaries are 64KB aligned
- ASLR only affects the top 16 bits
- Overwriting the low 16 bits of a pointer will shift it by up to 64KB to a known location inside the same DLL
- Exploitation is vulnerability specific

Memory disclosure

- If we can read memory from the process, we can bypass ASLR
- Even a single return address from the stack is enough to get the base of a DLL
- DEP can be bypassed with return oriented shellcode

ASLR entropy attacks

- ASLR on Windows provides only 8 bits of entropy
- If we can try an exploit 256 times, we can bypass ASLR by guessing the base address of a DLL
- DEP can be bypassed with return oriented shellcode

Virtual shellcode

- We can write our shellcode as a Java applet and use memory corruption to disable the Java bytecode verification
- No need to worry about DEP at all!
- Can be achieved by overwriting a single byte in the JVM
- ASLR makes it harder to find the JVM, but other attacks of this kind might be possible

Corrupting application data

- We can change the behavior of a program by corrupting its data without modifying the control flow
- Stack and heap overflows can corrupt data
- How do we find the right data to overwrite?

Directions for future research

1. Are there new classes of C or C++ vulnerabilities that lead to memory disclosure?

   Are there more general ways to get memory disclosure from the currently known vulnerability classes?

Directions for future research

2. Can we automate any of the manual analysis work required to exploit partial overwrites or data corruption vulnerabilities?

Directions for future research

3. Can we use static or dynamic binary analysis to improve our control over the memory layout of a process?
   - How do we find all data in memory that is used by an authentication function?
   - How do we ensure a heap block containing such data is allocated next to a heap block I can overflow?
   - How do we get control over the value of an stack or heap variable that is used before initialization?

Part IV

Conclusion

Conclusion

- Will the exploit mitigations really stop exploitation?
- We need a more research in this area
- Exploitation problems are hard
- If all else fails, web vulnerabilities will always be there!

Questions?

alex@sotirov.net