Exploitation Chronomancy

Temporal Return Addresses

skape

toorcon, 2005

Part I

Introduction

Who am I?

- Matt Miller (mmiller@hick.org)

Who am I?

- Matt Miller (mmiller@hick.org)

- Software developer
- Security enthusiast
- Metasploit contributor
- Win32 HIPS researcher
- Professional thumb wrestler

Plan of attack

- A brief background on return addresses

Plan of attack

- A brief background on return addresses
- Description and analysis of temporal addresses
  - What they are

Plan of attack

- A brief background on return addresses
- Description and analysis of temporal addresses
  - What they are
  - Why they're useful

Plan of attack

- A brief background on return addresses
- Description and analysis of temporal addresses
  - What they are
  - Why they're useful
  - How to find them

Plan of attack

- A brief background on return addresses
- Description and analysis of temporal addresses
  - What they are
  - Why they're useful
  - How to find them
  - How to use them

Plan of attack

- A brief background on return addresses
- Description and analysis of temporal addresses
  - What they are
  - Why they're useful
  - How to find them
  - How to use them
- Temporal return addresses in action
  - Windows NT `SharedUserData`

What are return addresses?

- What do I mean by *return address*?

What are return addresses?

- What do I mean by *return address*?
- An address that results in direct or indirect control of execution flow
  - Not limited to stack-based overflow of the return address

What are return addresses?

- What do I mean by *return address*?
- An address that results in direct or indirect control of execution flow
  - Not limited to stack-based overflow of the return address
- Direct
  - An address of shellcode on the stack

What are return addresses?

- What do I mean by *return address*?
- An address that results in direct or indirect control of execution flow
  - Not limited to stack-based overflow of the return address
- Direct
  - An address of shellcode on the stack
- Indirect
  - An address of a `jmp esp` instruction
  - A heap-based address stored in DTORs or elsewhere

What types of return addresses do people use?

- On Windows...
  - System and application DLLs with useful opcodes
    - `jmp esp`
    - `pop/pop/ret`

What types of return addresses do people use?

- On Windows...
  - System and application DLLs with useful opcodes
    - `jmp esp`
    - `pop/pop/ret`
  - Stack addresses often times have null bytes or are unpredictable

What types of return addresses do people use?

- On Windows...
  - System and application DLLs with useful opcodes
    - `jmp esp`
    - `pop/pop/ret`
  - Stack addresses often times have null bytes or are unpredictable
- On UNIX derivatives
  - Stack/heap addresses pointing directly to the shellcode

What types of return addresses do people use?

- On Windows...
  - System and application DLLs with useful opcodes
    - `jmp esp`
    - `pop/pop/ret`
  - Stack addresses often times have null bytes or are unpredictable
- On UNIX derivatives
  - Stack/heap addresses pointing directly to the shellcode
  - It's rare to see bouncing off useful opcodes in shared libraries

What types of return addresses do people use?

- On Windows...
  - System and application DLLs with useful opcodes
    - `jmp esp`
    - `pop/pop/ret`
  - Stack addresses often times have null bytes or are unpredictable
- On UNIX derivatives
  - Stack/heap addresses pointing directly to the shellcode
  - It's rare to see bouncing off useful opcodes in shared libraries
- It is very uncommon, but not unheard of, to have an addressless exploit

How reliable are most return addresses?

- Most exploits make assumptions about address space layout
- If the address space is different, the exploit will fail

How reliable are most return addresses?

- Most exploits make assumptions about address space layout
- If the address space is different, the exploit will fail

- Often times, address assumptions are not portable between OS and application revisions
  - `ws2help.dll` is good, but addresses aren't portable between NT, 2000, XP, and 2003

How reliable are most return addresses?

- Most exploits make assumptions about address space layout
- If the address space is different, the exploit will fail

- Often times, address assumptions are not portable between OS and application revisions
  - `ws2help.dll` is good, but addresses aren't portable between NT, 2000, XP, and 2003
- This forces exploits to have version specific targets

Dealing with version-specific return addresses

- Some exploits can be made universal even with version specific addresses
    - Metasploit's RPC DCOM exploit

Dealing with version-specific return addresses

- Some exploits can be made universal even with version specific addresses
  - Metasploit's RPC DCOM exploit
- Sometimes the OS/app version can be reliably determined
  - Especially common in browser-based exploits, among others

Dealing with version-specific return addresses

- Some exploits can be made universal even with version specific addresses
  - Metasploit's RPC DCOM exploit
- Sometimes the OS/app version can be reliably determined
  - Especially common in browser-based exploits, among others
- In other cases, target selection is a shot in the dark
  - Using the wrong target can result in a lost opportunity

Dealing with version-specific return addresses

- Some exploits can be made universal even with version specific addresses
  - Metasploit's RPC DCOM exploit
- Sometimes the OS/app version can be reliably determined
  - Especially common in browser-based exploits, among others
- In other cases, target selection is a shot in the dark
  - Using the wrong target can result in a lost opportunity
- Is there any way we can improve this?

Dealing with version-specific return addresses

- Some exploits can be made universal even with version specific addresses
  - Metasploit's RPC DCOM exploit
- Sometimes the OS/app version can be reliably determined
  - Especially common in browser-based exploits, among others

- In other cases, target selection is a shot in the dark
  - Using the wrong target can result in a lost opportunity

- Is there any way we can improve this?
- We'll see :)

Can moving targets be useful?

- A process' address space is constantly changing

Can moving targets be useful?

- A process' address space is constantly changing
- Thread stacks are always in a state of flux

Can moving targets be useful?

- A process' address space is constantly changing

- Thread stacks are always in a state of flux
- Heap regions are changed as more data is allocated and freed

Can moving targets be useful?

- A process' address space is constantly changing

- Thread stacks are always in a state of flux
- Heap regions are changed as more data is allocated and freed
- Files are mapped into memory and subsequently unmapped

Can moving targets be useful?

- A process' address space is constantly changing

- Thread stacks are always in a state of flux
- Heap regions are changed as more data is allocated and freed
- Files are mapped into memory and subsequently unmapped
- DLLs are loaded and unloaded as necessary

Can moving targets be useful?

- A process' address space is constantly changing

- Thread stacks are always in a state of flux
- Heap regions are changed as more data is allocated and freed
- Files are mapped into memory and subsequently unmapped
- DLLs are loaded and unloaded as necessary

- When searching for viable return addresses, only static regions are analyzed
  - Typically limited to loaded images and possibly stacks
  - A few other regions are sometimes of use too, such as PEB/TEB

Can moving targets be useful?

- A process' address space is constantly changing

- Thread stacks are always in a state of flux
- Heap regions are changed as more data is allocated and freed
- Files are mapped into memory and subsequently unmapped
- DLLs are loaded and unloaded as necessary

- When searching for viable return addresses, only static regions are analyzed
  - Typically limited to loaded images and possibly stacks
  - A few other regions are sometimes of use too, such as PEB/TEB
- Are we missing anything important by ignoring non-static regions?

Can moving targets be useful?

- Dynamic regions of memory can contain useful opcodes, just like static regions
  - A pointer stored in the heap can be composed of a viable opcode
  - An integer stored in a variable can be composed of a viable opcode

Can moving targets be useful?

- Dynamic regions of memory can contain useful opcodes, just like static regions
  - A pointer stored in the heap can be composed of a viable opcode
  - An integer stored in a variable can be composed of a viable opcode
- The problem is that their state is inherently transient

Can moving targets be useful?

- Dynamic regions of memory can contain useful opcodes, just like static regions
  - A pointer stored in the heap can be composed of a viable opcode
  - An integer stored in a variable can be composed of a viable opcode

- The problem is that their state is inherently transient
- However, transient states can sometimes be predicted

Can moving targets be useful?

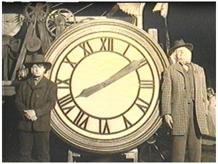
- Dynamic regions of memory can contain useful opcodes, just like static regions
  - A pointer stored in the heap can be composed of a viable opcode
  - An integer stored in a variable can be composed of a viable opcode

- The problem is that their state is inherently transient
- However, transient states can sometimes be predicted
- A good example of this can be seen in timer variables
  - I'll refer to them as *temporal addresses*

Part II

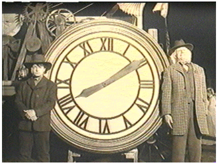
Temporal Addresses

Temporal addresses



- So just what is a temporal address, anyway?

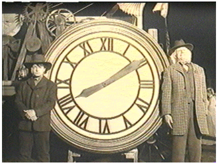
Temporal addresses



- So just what is a temporal address, anyway?

```
time_t foo = time(NULL);
```

Temporal addresses

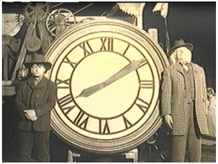


- So just what is a temporal address, anyway?

  ```
  time_t foo = time(NULL);
  ```

- It's a location in memory that contains timer state
  - The number of seconds since Jan 1, 1970
  - The number of seconds since a program started

Temporal addresses



- So just what is a temporal address, anyway?

```
time_t foo = time(NULL);
```

- It's a location in memory that contains timer state
  - The number of seconds since Jan 1, 1970
  - The number of seconds since a program started
- All temporal addresses have three basic properties

Temporal address properties

Capacity

- The maximum size of a temporal address' contents
- This limits the amount timer state it can hold

Temporal address properties

Capacity

- The maximum size of a temporal address' contents
- This limits the amount timer state it can hold

Period

- How often the timer state is updated

Temporal address properties

Capacity

- The maximum size of a temporal address' contents
- This limits the amount timer state it can hold

Period

- How often the timer state is updated

Scale

- The unit of measure associated with the timer
  - Number of seconds since epoch 1970
  - Number of seconds since epoch 1601
  - Counter from program start

Why are temporal addresses useful?

- Timer state is just a series of bytes in a certain order
- Knowing the three properties of a temporal address is handy

Why are temporal addresses useful?

- Timer state is just a series of bytes in a certain order
- Knowing the three properties of a temporal address is handy
- It means you can predict two things
  - When certain byte combinations will occur
  - How long those byte combinations will last

Why are temporal addresses useful?

- Timer state is just a series of bytes in a certain order
- Knowing the three properties of a temporal address is handy
- It means you can predict two things
  - When certain byte combinations will occur
  - How long those byte combinations will last

- This makes temporal addresses potentially useful as return addresses
- All we need to know is when useful byte combinations will occur

An example of a temporal address

Example (little endian)

- Addresss: `0x01462004`
- Capacity: 4 bytes
- Period: 1 second
- Scale: Seconds since epoch (1970)

An example of a temporal address

Example (little endian)

- Addresss: `0x01462004`
- Capacity: 4 bytes
- Period: 1 second
- Scale: Seconds since epoch (1970)

Analysis

- Let's say the temporal state reaches 1136808960 seconds

An example of a temporal address

Example (little endian)

- Addresss: `0x01462004`
- Capacity: 4 bytes
- Period: 1 second
- Scale: Seconds since epoch (1970)

Analysis

- Let's say the temporal state reaches 1136808960 seconds
- This is equivalent to `0x43c25400`

An example of a temporal address

Example (little endian)

- Addresss: `0x01462004`
- Capacity: 4 bytes
- Period: 1 second
- Scale: Seconds since epoch (1970)

Analysis

- Let's say the temporal state reaches 1136808960 seconds
- This is equivalent to `0x43c25400`
- `0x54 0xc2` is equivalent to `push esp / retn`

An example of a temporal address

Example (little endian)

- Addresss: `0x01462004`
- Capacity: 4 bytes
- Period: 1 second
- Scale: Seconds since epoch (1970)

Analysis

- Let's say the temporal state reaches 1136808960 seconds
- This is equivalent to `0x43c25400`
- `0x54 0xc2` is equivalent to `push esp / retn`
- This means on Monday, Jan. 09, 2006 at 4:16 pm, `0x01462005` can be used as a universal `esp => eip` instruction

An example of a temporal address

Example (little endian)

- Addresss: `0x01462004`
- Capacity: 4 bytes
- Period: 1 second
- Scale: Seconds since epoch (1970)

Analysis

- Let's say the temporal state reaches 1136808960 seconds
- This is equivalent to `0x43c25400`
- `0x54 0xc2` is equivalent to `push esp / retn`
- This means on Monday, Jan. 09, 2006 at 4:16 pm, `0x01462005` can be used as a universal `esp => eip` instruction
- It can only be used for 4 minutes and 16 seconds, though

Locating temporal addresses

- We’ve seen how temporal addresses can be useful
- But how do we go about locating them?

Locating temporal addresses

- We've seen how temporal addresses can be useful
- But how do we go about locating them?

- We know they all have a capacity, period, and scale
- So how can we use that to identify them?

Locating temporal addresses

- We've seen how temporal addresses can be useful
- But how do we go about locating them?

- We know they all have a capacity, period, and scale
- So how can we use that to identify them?
- There are a few approaches

Locating temporal addresses

- We've seen how temporal addresses can be useful
- But how do we go about locating them?

- We know they all have a capacity, period, and scale
- So how can we use that to identify them?
- There are a few approaches

- Manually analyze a process' address space

Locating temporal addresses

- We've seen how temporal addresses can be useful
- But how do we go about locating them?

- We know they all have a capacity, period, and scale
- So how can we use that to identify them?
- There are a few approaches

- Manually analyze a process' address space
- Breakpoint on timer-related functions and see where the output is stored

Locating temporal addresses

- We’ve seen how temporal addresses can be useful
- But how do we go about locating them?

- We know they all have a capacity, period, and scale
- So how can we use that to identify them?
- There are a few approaches

- Manually analyze a process’ address space
- Breakpoint on timer-related functions and see where the output is stored
- Use a program to compare address space differences over time to find patterns

Locating temporal addresses

- We've seen how temporal addresses can be useful
- But how do we go about locating them?

- We know they all have a capacity, period, and scale
- So how can we use that to identify them?
- There are a few approaches

- Manually analyze a process' address space
- Breakpoint on timer-related functions and see where the output is stored
- Use a program to compare address space differences over time to find patterns

- Let's focus on the latter

Locating temporal addresses

- The most automatable way is through diffing

Locating temporal addresses

- The most automatable way is through diffing

- A process' address space is polled n times
- Each polling cycle is spread apart by t seconds

Locating temporal addresses

- The most automatable way is through diffing

- A process' address space is polled *n* times
- Each polling cycle is spread apart by *t* seconds

- Memory contents are diffed each time
- Locations that change at a constant rate are flagged

Locating temporal addresses

- The most automatable way is through diffing

- A process' address space is polled n times
- Each polling cycle is spread apart by t seconds

- Memory contents are diffed each time
- Locations that change at a constant rate are flagged

- Once finished, each flagged address can have its capacity, period, and scale calculated

Locating temporal addresses

- The most automatable way is through diffing

- A process' address space is polled *n* times
- Each polling cycle is spread apart by *t* seconds

- Memory contents are diffed each time
- Locations that change at a constant rate are flagged

- Once finished, each flagged address can have its capacity, period, and scale calculated

- If an address had its contents incremented by 5000 each cycle and *t* was 5 seconds
  - The period could be between 1 second and 1 millisecond

Example of locating temporal addresses

```
C:\>telescope 2620
[*] Attaching to process 2620 (5 polling cycles)...
[*] Polling address space........

Temporal address locations:

0x0012FE88 [Size=4, Scale=Counter, Period=1 sec]
0x0012FF7C [Size=4, Scale=Epoch (1970), Period=1 sec]
0x7FFE0000 [Size=4, Scale=Counter, Period=600 msec]
0x7FFE0014 [Size=8, Scale=Epoch (1601), Period=100 nsec]
```

Determining temporal address byte durations

- Finding a temporal address is only the first step
- Next, we need to calculate how long it takes for each byte to change

Determining temporal address byte durations

- Finding a temporal address is only the first step
- Next, we need to calculate how long it takes for each byte to change

- Each byte has 256 combinations (`0x00 - 0xff`)
- Calculating iterations between change of each byte index x is described as

$$duration(x) = 256^x$$

Determining temporal address byte durations

- Finding a temporal address is only the first step
- Next, we need to calculate how long it takes for each byte to change

- Each byte has 256 combinations (`0x00 - 0xff`)
- Calculating iterations between change of each byte index x is described as

$$duration(x) = 256^x$$

- Using the temporal address period, we can calculate how long it takes for each byte to change

$$tosec(x) = duration(x)/period$$

Determining temporal address byte durations

- Finding a temporal address is only the first step
- Next, we need to calculate how long it takes for each byte to change

- Each byte has 256 combinations (`0x00 - 0xff`)
- Calculating iterations between change of each byte index x is described as

$$duration(x) = 256^x$$

- Using the temporal address period, we can calculate how long it takes for each byte to change

$$tosec(x) = duration(x)/period$$

- These calculations tell us the byte index to start our search at

Example temporal address byte durations

Byte durations for a 4 byte temporal address that updates every second

```
$ ./chronomancer.rb -a 4-1s-1970 -i

Interval of time it takes to change each byte:

0: 1 sec
1: 4 mins 16 secs
2: 18 hours 12 mins 16 secs
3: 194 days 4 hours 20 mins 16 secs
```

Example temporal address byte durations

Byte durations for a 4 byte temporal address that updates every second

```
$ ./chronomancer.rb -a 4-1s-1970 -i

Interval of time it takes to change each byte:

0: 1 sec
1: 4 mins 16 secs
2: 18 hours 12 mins 16 secs
3: 194 days 4 hours 20 mins 16 secs
```

Our best bet would be to start viable opcode searches at byte index 1

Calculating viable opcode windows

- Let's review what we've got so far
  - A temporal address with a capacity, period, and scale
  - The duration of each byte within the temporal address

Calculating viable opcode windows

- Let's review what we've got so far
  - A temporal address with a capacity, period, and scale
  - The duration of each byte within the temporal address
- Now it's time to predict the future!

Calculating viable opcode windows

- Let's review what we've got so far
  - A temporal address with a capacity, period, and scale
  - The duration of each byte within the temporal address
- Now it's time to predict the future!
- First we need to define our viable opcode set
  - `jmp esp`
  - `push esp, ret`
  - etc

Calculating viable opcode windows

- Let's review what we've got so far
  - A temporal address with a capacity, period, and scale
  - The duration of each byte within the temporal address
- Now it's time to predict the future!
- First we need to define our viable opcode set
  - `jmp esp`
  - `push esp, ret`
  - etc
- From there we can create all the viable opcode permutations

Calculating viable opcode permutations

- Pretty simple algorithm
- Plug the viable opcode bytes into each byte offset starting at a predetermined byte index

Calculating viable opcode permutations

- Pretty simple algorithm
- Plug the viable opcode bytes into each byte offset starting at a predetermined byte index

- If we had a temporal address with a 1 second period, we'd do...
  - `0xff` at byte index 1, `0xe4` at byte index 2
  - `0xff` at byte index 2, `0xe4` at byte index 3

Calculating viable opcode permutations

- Pretty simple algorithm
- Plug the viable opcode bytes into each byte offset starting at a predetermined byte index
- If we had a temporal address with a 1 second period, we'd do...
  - `0xff` at byte index 1, `0xe4` at byte index 2
  - `0xff` at byte index 2, `0xe4` at byte index 3
- From there it's necessary to generate all the byte combinations that could occur surrounding the viable opcode bytes
- The result is all the possible timer states containing the viable opcode bytes

Calculating viable opcode permutations

- Pretty simple algorithm
- Plug the viable opcode bytes into each byte offset starting at a predetermined byte index

- If we had a temporal address with a 1 second period, we'd do...
  - `0xff` at byte index 1, `0xe4` at byte index 2
  - `0xff` at byte index 2, `0xe4` at byte index 3

- From there it's necessary to generate all the byte combinations that could occur surrounding the viable opcode bytes
- The result is all the possible timer states containing the viable opcode bytes

- After all the permutations are calculated, all we need to do is figure out when to strike

Part III

Picking a Time to Strike

Figuring out when to strike

- Knowing the location and future states of temporal addresses is not enough

Figuring out when to strike

- Knowing the location and future states of temporal addresses is not enough
- In order to use them, timing information must be determined

Figuring out when to strike

- Knowing the location and future states of temporal addresses is not enough
- In order to use them, timing information must be determined
- If the scale is measuring system time, we need to know the system time
- If the scale is measuring time since program start, we need to know when the program started

Figuring out when to strike

- Knowing the location and future states of temporal addresses is not enough
- In order to use them, timing information must be determined
- If the scale is measuring system time, we need to know the system time
- If the scale is measuring time since program start, we need to know when the program started

- The latter may be infeasible
- But determining system time is not

Determining remote system time

- DCERPC SrvSvc NetrRemoteTOD

```
⊟ Microsoft Server Service, NetrRemoteTOD
    Operation: NetrRemoteTOD (28)
  ⊟ Time of day
      Referent ID: 0x001628b8
      Elapsed: 1123299129
      msecs: 1399879906
      Hours: 3
      Mins: 32
      Secs: 9
      Hunds: 27
      Timezone: 300
      Tinterval: 310
      Day: 6
      Month: 8
      Year: 2005
      Weekday: 6
```

Determining remote system time

- If the remote box is a web server, the HTTP date header can be used

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Date: Sat, 06 Aug 2005 03:38:06 GMT\r\n
  Server: Microsoft-IIS/6.0\r\n
  Last-Modified: Mon, 24 Mar 2003 07:11:10 GMT\r\n
  ETag: "2f00a0-acd-3e7eaf8e"\r\n
  Accept-Ranges: bytes\r\n
  Content-Length: 2765\r\n
  Connection: close\r\n
  Content-Type: text/html\r\n
  \r\n
```

Determining remote system time

- Lots of other ways exist...

Determining remote system time

- Lots of other ways exist...
- ICMP Timestamp

Determining remote system time

- Lots of other ways exist...

- ICMP Timestamp
- IP Timestamp

Determining remote system time

- Lots of other ways exist...

- ICMP Timestamp
- IP Timestamp
- IRC CTCP TIME

Determining remote system time

- Lots of other ways exist...

- ICMP Timestamp
- IP Timestamp
- IRC CTCP TIME
- SSL negotiations

Determining remote system time

- Lots of other ways exist...

- ICMP Timestamp
- IP Timestamp
- IRC CTCP TIME
- SSL negotiations

- And the list goes on

Part IV

Case Study: Windows NT SharedUserData

What is SharedUserData

- Shared region of memory
- Found in every win32 process
- Located at `0x7ffe0000` in every version of Windows NT+
- Executable up until XPSP2 + PAE
- Biggest draw back is that it contains a NULL byte
- But why's this related to this presentation?

What is SharedUserData

- Shared region of memory
- Found in every win32 process
- Located at `0x7ffe0000` in every version of Windows NT+
- Executable up until XPSP2 + PAE
- Biggest draw back is that it contains a NULL byte
- But why's this related to this presentation?

- Because it contains temporal addresses

The SharedUserData data structure

```
0:000> dt _KUSER_SHARED_DATA
   +0x000 TickCountLow       : Uint4B
   +0x004 TickCountMultiplier : Uint4B
   +0x008 InterruptTime      : _KSYSTEM_TIME
   +0x014 SystemTime         : _KSYSTEM_TIME
   +0x020 TimeZoneBias       : _KSYSTEM_TIME
   +0x02c ImageNumberLow     : Uint2B
   ...
```

Looking at the first few bytes of `SharedUserData` is interesting

```
0:000> dd 0x7ffe0000 L8
7ffe0000  055d7525 0fa00000 93fd5902 00000cca
7ffe0010  00000cca a78f0b48 01c59a46 01c59a46
0:000> dd 0x7ffe0000 L8
7ffe0000  055d7558 0fa00000 9477d5d2 00000cca
7ffe0010  00000cca a808a336 01c59a46 01c59a46
0:000> dd 0x7ffe0000 L8
7ffe0000  055d7587 0fa00000 94e80a7e 00000cca
7ffe0010  00000cca a878b1bc 01c59a46 01c59a46
```

Temporal addresses found in SharedUserData

TickCountLow

- Address: `0x7ffe0000`
- Capacity: 4 bytes
- Period: Variable
- Scale: Milliseconds since boot

Temporal addresses found in SharedUserData

TickCountLow

- Address: `0x7ffe0000`
- Capacity: 4 bytes
- Period: Variable
- Scale: Milliseconds since boot

InterruptTime

- Address: `0x7ffe0008`
- Capacity: 8 bytes
- Period: Variable
- Scale: 100ns time processing interrupts

Temporal addresses found in SharedUserData

TickCountLow

- Address: `0x7ffe0000`
- Capacity: 4 bytes
- Period: Variable
- Scale: Milliseconds since boot

InterruptTime

- Address: `0x7ffe0008`
- Capacity: 8 bytes
- Period: Variable
- Scale: 100ns time processing interrupts

SystemTime

- Address: `0x7ffe0014`
- Capacity: 8 bytes
- Period: 100 nanoseconds
- Scale: 100ns intervals since epoch 1601

SystemTime rocks

- SystemTime stores the count of 100ns intervals since 1601
  - Note that it does not appear to account for daylight savings time

SystemTime rocks

- SystemTime stores the count of 100ns intervals since 1601
  - Note that it does not appear to account for daylight savings time
- At a structural level it's a `KSYSTEM_TIME` structure

```
0:000> dt _KSYSTEM_TIME
   +0x000 LowPart          : Uint4B
   +0x004 High1Time        : Int4B
   +0x008 High2Time        : Int4B
```

- Let's see how we can abuse this

Taking advantage of the SystemTime attribute

First we need to calculate the byte durations based on the period

```
$ ./chronomancer.rb -a 8-100ns-1601 -i

Interval of time it takes to change each byte:

0: <1 sec
1: <1 sec
2: <1 sec
3: 1 sec
4: 7 mins 9 secs
5: 1 day 6 hours 32 mins 31 secs
6: 325 days 18 hours 44 mins 57 secs
7: 228 years 179 days 23 hours 50 mins 3 secs
```

Taking advantage of the SystemTime attribute

First we need to calculate the byte durations based on the period

```
$ ./chronomancer.rb -a 8-100ns-1601 -i

Interval of time it takes to change each byte:

0: <1 sec
1: <1 sec
2: <1 sec
3: 1 sec
4: 7 mins 9 secs
5: 1 day 6 hours 32 mins 31 secs
6: 325 days 18 hours 44 mins 57 secs
7: 228 years 179 days 23 hours 50 mins 3 secs
```

Looks like we should start at byte index 4, that would at least give us a 7 minute window

Generating the permutations

- The final step is to generate permutations

Generating the permutations

- The final step is to generate permutations
- We could do this manually...

Generating the permutations

- The final step is to generate permutations

- We could do this manually...
- Or we could use a script :)

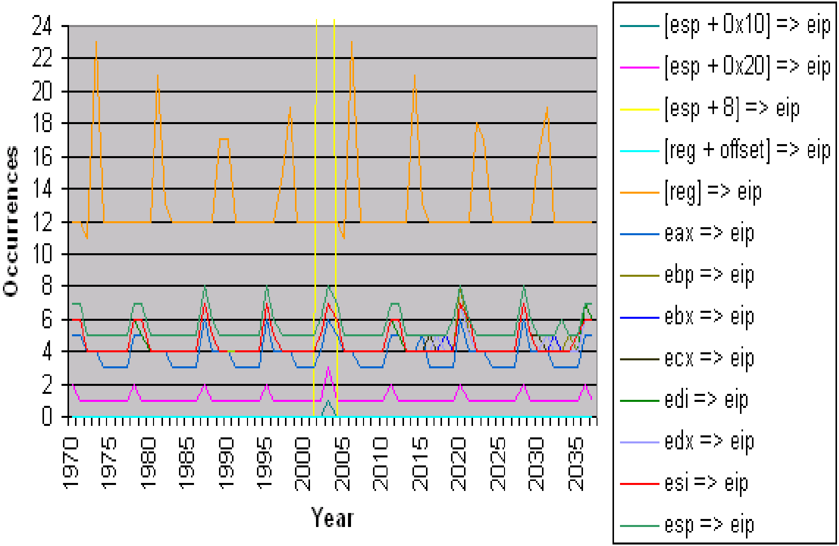
```
$ ./chronomancer.rb -a 8-100ns-1601

...
1/1970,1807823,Wed Jan 21 16:10:23 CST 1970,
    0000000050c29d01,eax => eip,7 mins 9 secs
1/1970,1808252,Wed Jan 21 16:17:32 CST 1970,
    0000000051c29d01,ecx => eip,7 mins 9 secs
1/1970,1808682,Wed Jan 21 16:24:42 CST 1970,
    0000000052c29d01,edx => eip,7 mins 9 secs
1/1970,1809111,Wed Jan 21 16:31:51 CST 1970,
    0000000053c29d01,ebx => eip,7 mins 9 secs
1/1970,1809541,Wed Jan 21 16:39:01 CST 1970,
    0000000054c29d01,esp => eip,7 mins 9 secs
...
```

Upcoming viable opcode windows for SystemTime

Watch out in September of this year!

Date	Opcode Group
Sun Sep 25 22:08:50 CDT 2005	eax => eip
Sun Sep 25 22:15:59 CDT 2005	ecx => eip
Sun Sep 25 22:23:09 CDT 2005	edx => eip
Sun Sep 25 22:30:18 CDT 2005	ebx => eip
Sun Sep 25 22:37:28 CDT 2005	esp => eip
Sun Sep 25 22:44:37 CDT 2005	ebp => eip
Sun Sep 25 22:51:47 CDT 2005	esi => eip
Sun Sep 25 22:58:56 CDT 2005	edi => eip

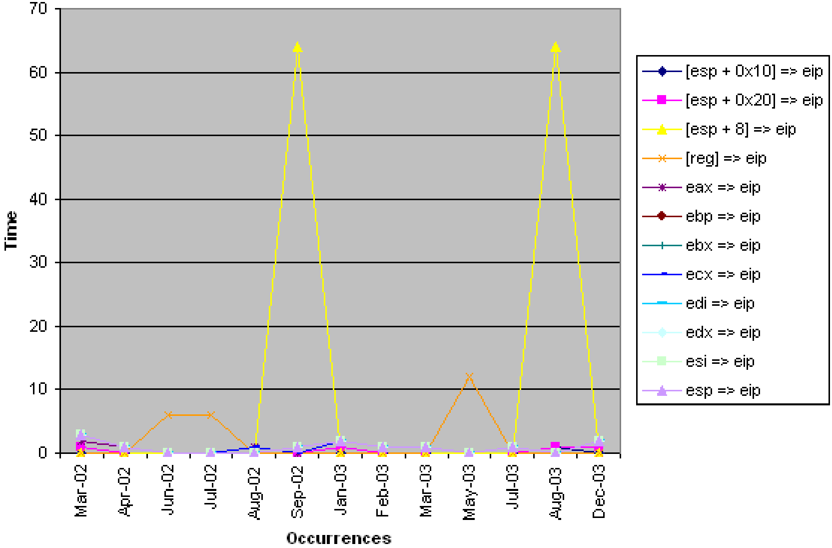
Plotting viable opcode windows for SystemTime



What's with the [esp + 8] spikes?

- In 2002 and 2003, SystemTime had a jump in occurrences of `[esp + 8] => eip` combinations
  - `[esp + 8]` is equivalent to `pop/pop/ret`
- It's too bad this technique wasn't applied then!
- Never again in our lifetime will that spike recur

The [esp + 8] spikes



Part V

Conclusion

So how probable is this anyway?

- In general, this technique isn’t very feasible
- Viable opcode windows are usually pretty far apart
- It might not always be possible to get system timing information
- The list goes on...

So how probable is this anyway?

- In general, this technique isn't very feasible
- Viable opcode windows are usually pretty far apart
- It might not always be possible to get system timing information
- The list goes on...

- But what if you compromised an NTP server?
- This would give you control over things `SystemTime`
- And you would automatically know what hosts to target
- That doesn't seem too infeasible...

Conclusion

- Check out the uninformed paper for a more detailed explanation
  - `http://www.uninformed.org`
- Includes code for...
  - Locating temporal addresses on win32 (telescope.c)
  - Calculating viable opcode windows and byte durations (chronomancer)

Questions

Questions?