Temporal Return Addresses

Exploitation Chronomancy

skape
mmiller@hick.org

Last modified: 8/6/2005

Contents

Chapter 1

Foreword

Abstract: Nearly all existing exploitation vectors depend on some knowledge of a process' address space prior to an attack in order to gain meaningful control of execution flow. In cases where this is necessary, exploit authors generally make use of static addresses that may or may not be portable between various operating system and application revisions. This fact can make exploits unreliable depending on how well researched the static addresses were at the time that the exploit was implemented. In some cases, though, it may be possible to predict and make use of certain addresses in memory that do not have static contents. This document introduces the concept of *temporal addresses* and describes how they can be used, under certain circumstances, to make exploitation more reliable.

Thanks: The author would like to thank H D Moore, spoonm, thief, jhind, johnycsh, vlad902, warlord, trew, #vax, uninformed, and all the friends of nologin!

With that, on with the show...

Chapter 2

Introduction

A common impediment to the implementation of portable and reliable exploits is the location of a return address. It is often required that a specific instruction, such as a `jmp esp`, be located at a predictable location in memory so that control flow can be redirected into an attacker controlled buffer[1]. Many times, though, the locations of the instructions will vary between individual versions of an operating system, thus limiting an exploit to a set of version-specific targets that may or may not be directly determinable at attack time. In order to make an exploit independent of, or at least less dependent on, a target's operating system version, a shift in focus becomes necessary.

Through the blur of rhyme and reason an attacker might focus and realize that not all viable return addresses will exist indeterminably in a target process' address space. In fact, viable return addresses can be found in a transient state throughout the course of a program's execution. For instance, a pointer might be stored at a location in memory that happens to contain a viable two byte instruction somewhere within the bytes that compose the pointer's address. Alternatively, an integer value somewhere in memory could be initialized to a value that is equivalent to a viable instruction. In both cases, though, the contents and locations of the values will almost certainly be volatile and unpredictable, thus making them unsuitable for use as return addresses.

Fortunately, however, there does exist at least one condition that can lend itself well to portable exploitation that is bounded not by the operating system version the target is running on, but instead by a defined window of time. In a condition such as this, a timer of some sort must exist at a predictable location in memory that is known to be updated at a constant time interval, such as every second. The location in memory that the timer resides at is known as a *temporal address*.

[1]This scenario is more common on Windows, but applicable scenarios exist on UNIX derivatives as well

On top of this, it is also important for the attacker determine the scale of measurement the timer is operating on, such as whether or not it's measured in epoch time (from 1970 or 1601) or if it's simply acting as a counter. With these three elements identified, an attacker can attempt to predict the periods of time where a useful instruction can be found in the bytes that compose the future state of any timer in memory.

To help illustrate this, suppose an attacker is attempting to find a reliable location of a `jmp edi` instruction. The attacker knows that the program being exploited has a timer that holds the number of seconds since Jan. 1, 1970 at a predictable location in memory. By doing some analysis, the attacker could determine that on Wednesday July 27th, 2005 at 3:39:12PM CDT, a `jmp edi` could be found within any four byte timer that stores the number of seconds since 1970. The window of opportunity, however, would only last for 4 minutes and 16 seconds assuming the timer is updated every second.

By accounting for timing as a factor in the selection of return addresses, an attacker can be afforded options beyond those normally seen when the address space of a process is viewed as unchanging over time. In that light, this document is broken into three portions. First, the steps needed to find, analyze, and make use of temporal addresses will be explained. Second, upcoming viable opcode windows will be shown and explained along with methods that can be used to determine target time information prior to exploitation. Finally, examples of commonly occurring temporal addresses on Windows NT+ will be described and analyzed to provide real world examples of the subject of this document.

Before starting, though, it is important to understand some of the terminology that will be used, or perhaps abused, in the interest of conveying the concepts. The term *temporal address* is used to describe a location in memory that contains a timer of some sort. The term *opcode* is used interchangeably with the term *instruction* to convey the set of viable bytes that could partially compose a given temporal state. The term *update period* is used to describe the amount of time that it takes for the contents of a temporal address to change. Finally, the term *scale* is used to describe the unit of measure for a given temporal address.

Chapter 3

Locating Temporal Addresses

In order to make use of temporal addresses it is first necessary to devise a method of locating them. To begin this search it is necessary that one understand the attributes of a temporal address. All temporal addresses are defined as storing a time-associated counter that increments at a constant interval. For instance, an example would be a location in memory that stores the number of seconds since Jan. 1, 1970 that is incremented every second. As a more concrete definition, all time-associated counters found in memory are represented in terms of a scale (the unit of measure), an interval or period (how often they are updated), and have a maximum storage capacity (variable size). If any these parts are unknown or variant for a given memory location, it is impossible for an attacker to consistently leverage it for use as time-bounded return address because of the inability to predict the byte values at the location for a given period of time.

With the three major components of a temporal address identified (scale, period, and capacity), a program can be written to search through a process' address space with the goal of identifying regions of memory that are updated at a constant period. From there, a scale and capacity can be inferred based on an arbitrarily complex set of heuristics, the simplest of which can identify regions that are storing epoch time. It's important to note, though, that not all temporal addresses will have a scale that is measured as an absolute time period. Instead, a temporal address may simply store the number of seconds that have passed since the start of execution, among other scenarios. These temporal addresses are described as having a scale that is simply equivalent to their period and are for that reason referred to as *counters*.

To illustrate the feasibility of such a program, the author has implemented an algorithm that should be conceptually portable to all platforms, though the implementation itself is limited to Windows NT+. The approach taken by the author, at a high level, is to poll a process' address space multiple times with the intention of analyzing changes to the address space over time. In order to reduce the amount of memory that must be polled, the program is also designed to skip over regions that are backed against an image file or are otherwise inaccessible.

To accomplish this task, each polling cycle is designed to be separated by a constant (or nearly constant) time interval, such as 5 seconds[1]. The granularity of this period of time is measured in nanoseconds in order to support high resolution timers that may exist within the target process' address space. This allows the program to detect timers measured in nanoseconds, microseconds, milliseconds, and seconds. The purpose of the delay between polling cycles is to give temporal address candidates the ability to complete one or more update periods. As each polling cycle occurs, the program reads the contents of the target process' address space for a given region and caches it locally within the scanning process. This is necessary for the next phase.

After at least two polling cycles have completed, the program can compare the cached memory region differences between the most recent view of the target process' address space and the previous view. This is accomplished by walking through the contents of each cached memory region in four byte increments to see if there is any difference between the two views. If a temporal address exists, the contents of a the two views should have a difference that is no larger than the maximum period of time that occurred between the two polling cycles[2]. For instance, if the polling cycle period was 5 seconds, any portion of memory that changed by more than 5 seconds, 5000 milliseconds, or 5000000 microseconds is obviously not a temporal address candidate. To that point, any region of memory that didn't change at all is also most likely not a temporal address candidate, though it is possible that the region of memory simply has an update period that is longer than the polling cycle.

Once a memory location is identified that has a difference between the two views that is within or equal to the polling cycle period, the next step of analysis can begin. It's perfectly possible for memory locations that meet this requirement to not actually be timers, so further analysis is necessary to weed them out. At this point, though, memory locations such as these can be referred to as *temporal address candidates*. The next step is to attempt to determine the period of the temporal address candidate. This is accomplished by some rather silly, but functional, logic.

First, the delta between the polling cycles is calculated down to nanosecond granularity. In a best case scenario, the granularity of a polling cycle that is spaced apart by 5 seconds will be 5000000000 nanoseconds. It's not safe

[1]By increasing the interval between polling cycles the program can detect temporal addresses that have a larger update period

[2]It's important to remember that the maximum period can be conveyed down to nanosecond granularity

to assume this constant though, as thread scheduling and other non-constant parameters can affect the delta between polling cycles for a given memory region. The next step is to iteratively compare the difference between the two views to the current delta to see if the difference is greater than or equal to the current delta. If it is, it can be assumed that the difference is within the current unit of measure. If it's not, the current delta should be divided by 10 to progress to the next unit of measure. When broken down, the progressive transition in units of measurement is described in figure 3.1.

Delta	Measurement
1000000000	Nanoseconds
100000000	10 Nanoseconds
10000000	100 Nanoseconds
1000000	Microseconds
100000	10 Microseconds
10000	100 Microseconds
1000	Milliseconds
100	10 Milliseconds
10	100 Milliseconds
1	Seconds

Figure 3.1: Delta measurement reductions

Once a unit of measure for the update period is identified, the difference is divided by the current delta to produce the update period for a given temporal address candidate. For example, if the difference was 5 and the current delta was 5, the update period for the temporal address candidate would be 1 second (5 updates over the course of 5 seconds). With the update period identified, the next step is to attempt to determine the storage capacity of the temporal address candidate.

In this case, the author chose to take a shortcut, though there are most certainly better approaches that could be taken given sufficient interest. The author chose to assume that if the update period for a temporal address candidate was measured in nanoseconds, then it was almost certainly at least the size of a 64-bit integer[3]. On the other hand, all other update periods were assumed to imply a 32-bit integer[4].

With the temporal address candidate's storage capacity identified in terms of bytes, the next step is to identify the scale that the temporal address may be conveying (the timer's unit of measure). To accomplish this, the program calculates the number of seconds since 1970 and 1601 between the current time minus at least equal the polling cycle period and the current time itself. The temporal address candidate's current value (as stored in memory) is then converted to

[3] 8 bytes on x86

[4] 4 bytes on x86

seconds using the determined update period and then compared against the two epoch time ranges. If the candidate's converted current value is within either epoch time range then it can most likely be assumed that the temporal address candidates's scale is measured from epoch time, either from 1970 or 1601 depending on the range it was within. While this sort of comparison is rather simple, any other arbitrarily complex set of logic could be put into place to detect other types of time scales. In the event that none of the logic matches, the temporal address candidate is deemed to simply have a scale of a counter (as defined previously in this chapter).

Finally, with the period, scale, and capacity for the temporal address candidate identified, the only thing left is to check to see if the three components are equivalent to previously collected components for the given temporal address candidate. If they differ in orders of magnitude then it is probably safe to assume that the candidate is not actually a temporal address. On the other, consistent components between polling cycles for a temporal address candidate are almost a sure sign that it is indeed a temporal address.

When everything is said and done, the program should collect every temporal address in the target process that has an update period less than or equal to the polling cycle period. It should also have determined the scale and size of the temporal address. When run on Windows against a program that is storing the current epoch time since 1970 in seconds in a variable every second, the following output is displayed:

```
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]
```

This output tells us that the address of the variable that is storing the epoch time since 1970 can be found at `0x0012FF7C` and has an update period of one second. The other things that were found will be discussed later in this document.

3.1 Determining Per-byte Durations

Once the update period and size of a temporal address have been determined, it is possible to calculate the amount of time it takes to change each byte position in the temporal address. For instance, if a four byte temporal address with an update period of 1 second were found in memory, the first byte (or LSB) would change once every second, the second byte would change once every 256 seconds, the third byte would change once every 65536 seconds, and the fourth byte would change once every 16777216 seconds. The reason these properties are exhibited is because each byte position has 256 possibilities (`0x00` to `0xff` inclusive). This means that each byte position increases in duration by 256 to a given power. This can be described as shown in figure 3.1. Let x equal the byte index starting at zero for the LSB.

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

Figure 3.2: Period independent byte durations

The next step to take after determining period-specific byte durations is to convert the durations to a measure more aptly accessible assuming a period that is more granular than a second. For instance, figure 3.1 shows that if each byte duration is measured in 100 nanosecond intervals for an 8 byte temporal address, a conversion can be applied to convert from 100 nanosecond intervals for a byte duration to seconds.

$$tosec(x) = duration(x)/10^7$$

Figure 3.3: 100 nanosecond byte durations to seconds

This phase is especially important when it comes to calculating viable opcode windows because it is necessary to know for how long a viable opcode will exist which is directly dependent on the direction of the opcode byte closest to the LSB. This will be discussed in more detail in chapter 4.

Chapter 4

Calculating Viable Opcode Windows

Once a set of temporal addresses has been located, the next logical step is to attempt to calculate the windows of time that one or more viable opcodes can be found within the bytes of the temporal address. It is also just as important to calculate the duration of each byte within the temporal address. This is the type of information that is required in order to determine when a portion of a temporal address can be used as a return address for an exploit. The approach taken to accomplish this is to make use of the equations provided in the previous chapter for calculating the number of seconds it takes for each byte to change based on the update period for a given temporal address. By using the `tosec` function for each byte index, a table can be created as illustrated in figure 4.1 for a 100nanosecond 8 byte timer.

Byte Index	Seconds (ext)
0	0 (zero)
1	0 (zero)
2	0 (zero)
3	1 (1 sec)
4	429 (7 mins 9 secs)
5	109951 (1 day 6 hours 32 mins 31 secs)
6	28147497 (325 days 18 hours 44 mins 57 secs)
7	7205759403 (228 years 179 days 23 hours 50 mins 3 secs)

Figure 4.1: 8 byte 100ns per-byte durations in seconds

This shows that any opcodes starting at byte index 4 will have a 7 minute and 9 second window of time. The only thing left to do is figure out when to strike.

Chapter 5

Picking the Time to Strike

The time to attack is entirely dependent on both the update period of the temporal address and its scale. In most cases, temporal addresses that have a scale that is relative to an arbitrary date (such as 1970 or 1601) are the most useful because they can be predicted or determined with some degree of certainty. Regardless, a generalized approach can be used to determine projected time intervals where useful opcodes will occur.

To do this, it is first necessary to identify the set of instructions that could be useful for a given exploit, such as a `jmp esp`. Once identified, the next step is to break the instructions down into their raw opcodes, such as `0xff 0xe4` for `jmp esp`. After all the raw opcodes have been collected, it is then necessary to begin calculating the projected time intervals that the bytes will occur at. The method used to accomplish this is rather simple.

First, a starting byte index must be determined in terms of the lowest acceptable window of time that an exploit can use. In the case of a 100 nanosecond timer, the best byte index to start at would be byte index 4 considering all previous indexes have a duration of less than or equal to one second. The bytes that occur at index 4 have a 7 minute and 9 second duration, thus making them feasible for use. With the starting byte index determined, the next step is to create permutations of all subsequent opcode byte combinations. In simpler terms, this would mean producing all of the possible byte value combinations that contain the raw opcodes of a given instruction at a byte index equal to or greater than the starting byte index. To help visualize this, figure 5.1 provides a small sample of `jmp esp` byte combinations in relation to a 100 nanosecond timer.

Once all of the permutations have been generated, the next step is to convert them to meaingful absolute time representations. This is accomplished by converting all of the permutations, which represent past, future, or present

Byte combinations
00 00 00 00 ff e4 00 00
00 00 00 00 ff e4 01 00
00 00 00 00 ff e4 02 00
...
00 00 00 00 ff e4 47 04
00 00 00 00 ff e4 47 05
00 00 00 00 ff e4 47 06
...
00 00 00 00 00 ff e4 00
00 00 00 00 00 ff e4 01
00 00 00 00 00 ff e4 02

Figure 5.1: 8 byte 100ns `jmp esp` byte combinations

states of the temporal address, to seconds. For instance, one of the permutations for a `jmp esp` instruction found within the 64-bit 100nanosecond timer is `0x019de4ff00000000` (116500949249294300). Converting this to seconds is accomplished by doing:

$$11650094924 = trunc(116500949249294300/10^7)$$

This tells us the number of seconds that will have passed when the stars align to form this byte combination, but it does not convey the scale in which the seconds are measured, such as whether they are based from an absolute date (such as 1970 or 1601) or are simply acting as a timer. In this case, if the scale were defined as being the number of seconds since 1601, the total number of seconds could be adjusted to indicate the number of seconds that have occurred since 1970 by subtracting the constant number of seconds between 1970 and 1601:

$$5621324 = 11650094924 - 11644473600$$

This indicates that a total of `5621324` seconds will have passed since 1970 when `0xff` will be found at byte index 4 and `0xe4` will be found at byte index 5. The window of opportunity will be 7 minutes and 9 seconds after which point the `0xff` will become a `0x00`, the `0xe4` will become `0xe5`, and the instruction will no longer be usable. If `5621324` is converted to a printable date format based on the number of seconds since 1970, one can find that the date that this particular permutation will occur at is Fri Mar 06 19:28:44 CST 1970.

While it's now been shown that is perfectly possible to predict specific times in the past, present, and future that a given instruction or instructions can be found within a temporal address, such an ability is not useful without being able to predict or determine the state of the temporal address on a target computer at a specific moment in time. For instance, while an exploitation chronomancer knows that a `jmp esp` can be found on March 6th, 1970 at about 7:30 PM, it must also be known what the target machine has their system time set to down to a granularity of mere seconds, or at least minutes. While guessing is always an option, it is almost certainly going to be less fruitful than making use of existing tools and services that are more than willing to provide a would-be attacker with information about the current system time on a target machine. Some of the approaches that can be taken to gather this information will be discussed in the next section.

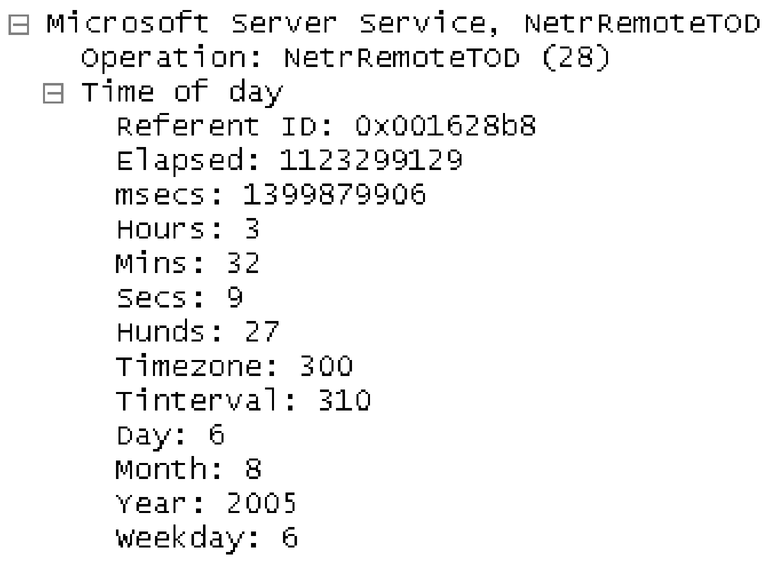
5.1 Determining System Time

There are a variety of techniques that can potentially be used to determine the system time of a target machine with varying degrees of accuracy. The techniques listed in this section are by no means all-encompassing but do serve as a good base. Each technique will be elaborated on in the following subsections.

5.1.1 DCERPC SrvSvc NetrRemoteTOD

One approach that can be taken to obtain very granular information about the current system time of a target machine is to use the SrvSvc's `NetrRemoteTOD` request. To transmit this request to a target machine a NULL session (or authenticated session) must be established using the standard `Session Setup AndX` SMB request. After that, a `Tree Connect AndX` to the `IPC$` share should be issued. From there, an `NT Create AndX` request can be issued on the `\srvsvc` named pipe. Once the request is handled successfully the file descriptor returned can be used for the DCERPC bind request to the SrvSvc's UUID. Finally, once the bind request has completed successfully, a `NetrRemoteTOD` request can be transacted over the named pipe using a `TransactNmPipe` request. The response to this request should contain very granular information, such as day, hour, minute, second, timezone, as well as other fields that are needed to determine the target machine's system time. Figure 5.2 shows a sample response.

This vector is very useful because it provides easy access to the complete state of a target machine's system time which in turn can be used to calculate the windows of time that a temporal address can be used during exploitation. The negatives to this approach is that it requires access to the SMB ports (either 139 or 445) which will most likely be inaccessible to an attacker.



```
⊟ 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
```

Figure 5.2: Example NetrRemoteTOD response

5.1.2 ICMP Timestamps

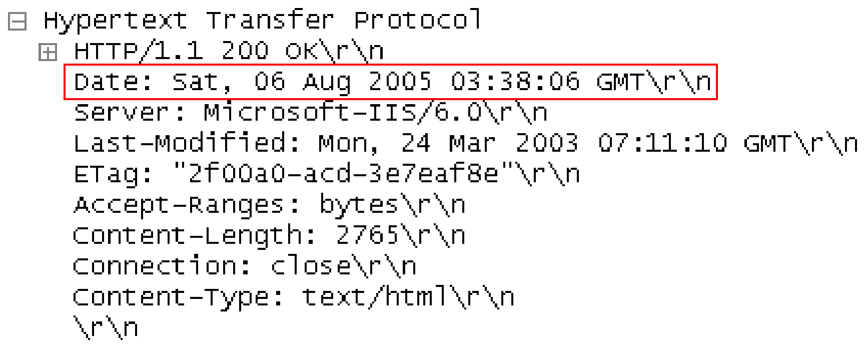
The ICMP `TIMESTAMP` request (13) can be used to obtain a machine's measurement of the number of milliseconds that have occurred since midnight UT. If an attacker can infer or assume that a target machine's system time is set to a specific date and timezone, it may be possible to calculate the absolute system time down to a millisecond resolution. This would satisfy the timing requirements and make it possible to make use of temporal addresses that have a scale that is measured from an absolute time. According to the RFC, though, if a system is unable to determine the number of milliseconds since UT then it can use another value capable of representing time (though it must set a high-order bit to indicate the non-standard value).

5.1.3 IP Timestamp Option

Like the ICMP `TIMESTAMP` request, IP also has a timestamp option (type 68) that measures the number of milliseconds since midnight UT. This could also be used to determine down to a millisecond resolution what the remote system's clock is set to. Since the measurement is the same, the limitations are the same as ICMP's `TIMESTAMP` request.

5.1.4 HTTP Server Date Header

In scenarios where a target machine is running an HTTP server, it may be possible to extract the system time by simply sending an HTTP request and checking to see if the response contains a date header or not. Figure 5.3 shows an example HTTP response that contains a date header.



```
⊟ 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
```

Figure 5.3: Example HTTP response

5.1.5 IRC CTCP TIME

Perhaps one of the more lame approaches to obtaining a target machine's time is by issuing a CTCP `TIME` request over IRC. This request is designed to instruct the responder to reply with a readable date string that is relative to the responder's system time. Unless spoofed, the response should be equivalent to the system time on the remote machine.

Chapter 6

Determining the Return Address

Once all the preliminary work of calculating all of the viable opcode windows has been completed and a target machine's system time has been determined, the final step is to select the next available window for a compatible opcode group. For instance, if the next window for a `jmp esp` equivalent instruction is Sun Sep 25 22:37:28 CDT 2005, then the byte index to the start of the `jmp esp` equivalent must be determined based on the permutation that was generated. In this case, the permutation that would have been generated (assuming a 100nanosecond period since 1601) is `0x01c5c25400000000`. This means that `jmp esp` equivalent is actually a `push esp, ret` which starts at byte index four. If the start of the temporal address was at `0x7ffe0014`, then the return address that should be used in order to get the `push esp, ret` to execute would be `0x7ffe0018`. This basic approach is common to all temporal addresses of varying capacity, period, and scale.

Chapter 7

Case Study: Windows NT SharedUserData

With all the generic background information out of the way, a real world practical use of this technique can be illustrated through an analysis of a region of memory that happens to be found in every process on Windows NT+. This region of memory is referred to as `SharedUserData` and has a backward compatible format for all versions of NT, though new fields have been appended over time. At present, the data structure that represents `SharedUserData` is `_KUSER_SHARED_DATA` which is defined as follows on Windows XP SP2:

```
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
   +0x02e ImageNumberHigh  : Uint2B
   +0x030 NtSystemRoot     : [260] Uint2B
   +0x238 MaxStackTraceDepth : Uint4B
   +0x23c CryptoExponent   : Uint4B
   +0x240 TimeZoneId       : Uint4B
   +0x244 Reserved2        : [8] Uint4B
   +0x264 NtProductType    : _NT_PRODUCT_TYPE
   +0x268 ProductTypeIsValid : UChar
   +0x26c NtMajorVersion   : Uint4B
   +0x270 NtMinorVersion   : Uint4B
   +0x274 ProcessorFeatures : [64] UChar
   +0x2b4 Reserved1        : Uint4B
   +0x2b8 Reserved3        : Uint4B
   +0x2bc TimeSlip         : Uint4B
   +0x2c0 AlternativeArchitecture : _ALTERNATIVE_ARCHITECTURE_TYPE
   +0x2c8 SystemExpirationDate : _LARGE_INTEGER
   +0x2d0 SuiteMask        : Uint4B
   +0x2d4 KdDebuggerEnabled : UChar
   +0x2d5 NXSupportPolicy  : UChar
   +0x2d8 ActiveConsoleId  : Uint4B
   +0x2dc DismountCount    : Uint4B
   +0x2e0 ComPlusPackage   : Uint4B
   +0x2e4 LastSystemRITEventTickCount : Uint4B
   +0x2e8 NumberOfPhysicalPages : Uint4B
   +0x2ec SafeBootMode     : UChar
   +0x2f0 TraceLogging     : Uint4B
   +0x2f8 TestRetInstruction : Uint8B
   +0x300 SystemCall       : Uint4B
   +0x304 SystemCallReturn : Uint4B
   +0x308 SystemCallPad    : [3] Uint8B
   +0x320 TickCount        : _KSYSTEM_TIME
   +0x320 TickCountQuad    : Uint8B
   +0x330 Cookie           : Uint4B
```

One of the purposes of SharedUserData is to provide processes with a global and consistent method of obtaining certain information that may be requested frequently, thus making it more efficient than having to incur the performance hit of a system call. Furthermore, as of Windows XP, SharedUserData acts as an indirect system call re-director such that the most optimized system call instructions can be used based on the current hardware's support, such as by using sysenter over the standard int 0x2e.

As can be seen right off the bat, SharedUserData contains a few fields that pertain to the timing of the current system. Furthermore, if one looks closely, it can be seen that these timer fields are actually updated constantly as would be expected for any timer variable:

```
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
```

The three timing-related fields of most interest are TickCountLow, InterruptTime, and SystemTime. These three fields will be explained individually later in this chapter. Prior to that, though, it is important to understand some of the properties of SharedUserData and why it is that it's quite useful when it comes to temporal addresses.

7.1 The Properties of SharedUserData

There are a number of important properties of SharedUserData, some of which make it useful in terms of temporal addresses and others that make it somewhat infeasible depending on the exploit or hardware support. As far as the properties that make it useful go, SharedUserData is located at a static address, 0x7ffe0000, in every version of Windows NT+. Furthermore, SharedUserData is mapped into every process. The reasons for this are that NTDLL, and most likely other 3rd party applications, have been compiled and built with the assumption that SharedUserData is located at a fixed address[1]. On top of that, SharedUserData is required to have a backward compatible data structure which means that the offsets of all existing attributes will never shift, although new attributes may be, and have been, appended to the end of the data structure. Lastly, there are a few products for Windows that implement some form of ASLR. Unfortunately for these products, SharedUserData cannot be feasibly randomized, or at least the author is not aware of any approaches that wouldn't have severe performance impacts.

On the negative side of the house, and perhaps one of the most limiting factors when it comes to making use of SharedUserData, is that it has a null byte located at byte index one. Depending on the vulnerability, it may or may not be possible to use an attribute within SharedUserData as a return address due to NULL byte restrictions. As of XP SP2 and 2003 Server SP1, SharedUserData is no longer marked as executable and will result in a DEP violation (if enabled) assuming the hardware supports PAE. While this is not very common yet, it is sure to become the norm over the course of time.

7.2 Locating Temporal Addresses

As seen previously in this document, using the telescope program on any Windows application will result in the same two (or three) timers being displayed:

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

[1]This is something many people are abusing these days when it comes to passing code from kernel-mode to user-mode

```
[*] Polling address space........

Temporal address locations:

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

Referring to the structure definition described at the beginning of this chapter, it is possible for one to determine which attribute each of these addresses is referring to. Each of these three attributes will be discussed in detail in the following sub-sections.

7.2.1 TickCountLow

The `TickCountLow` attribute is used, in combination with the `TickCountMultiplier`, to convey the number of milliseconds that have occurred since system boot. To calculate the number of milliseconds since system boot, the following equation is used:

$$T = shr(TickCountLow * TickCountMultiplier, 24)$$

This attribute is representative of a temporal address that has a *counter* scale. It starts an unknown time and increments at constant intervals. The biggest problem with this attribute are the intervals that it increases at. It's possible that two machines in the same room with different hardware will have different update periods for the `TickCountLow` attribute. This makes it less feasible to use as a temporal address because the update period cannot be readily predicted. On the other hand, it may be possible to determine the current uptime of the machine through TCP timestamps or some alternative mechanism, but without the ability to determine the update period, the `TickCountLow` attribute seems unusable.

This attribute is located at `0x7ffe0000` on all versions of Windows NT+.

7.2.2 InterruptTime

This attribute is used to store a 100 nanosecond timer starting at system boot that presumably counts the amount of time spent processing interrupts. The attribute itself is stored as a `_KSYSTEM_TIME` structure which is defined as:

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

Depending on the hardware a machine is running, the `InterruptTime`'s period may be exactly equal to 100 nanoseconds. However, testing has seemed to confirm that this is not always the case. Given this, both the update period and the scale of the `InterruptTime` attribute should be seen as limiting factors. This fact makes it less useful because it has the same limitations as the `TickCountLow` attribute. Specifically, without knowing when the system booted and when the counter started, or how much time has been spent processing interrupts, it is not possible to reliably predict when certain bytes will be at certain offsets. Furthermore, the machine would need to have been booted for a significant amount of time in order for some of the useful instructions to be feasibly found within the bytes that compose the timer.

This attribute is located at `0x7ffe0008` on all versions of Windows NT+.

7.2.3 SystemTime

The `SystemTime` attribute is by far the most useful attribute when it comes to its temporal address qualities. The attribute itself is a 100 nanosecond timer that is measured from Jan. 1, 1601 which is stored as a _KSYSTEM_TIME structure like the `InterruptTime` attribute[2]. This means that it has an update period of 100 nanoseconds and has a scale that measures from Jan. 1, 1601. The scale is also measured relative to the timezone that the machine is using (with the exclusion of daylight savings time). If an attacker is able to obtain information about the system time on a target machine, it may be possible to make use of the `SystemTime` attribute as a valid temporal address for exploitation purposes.

This attribute is located at `0x7ffe0014` on all versions of Windows NT+.

7.3 Calculating Viable Opcode Windows

After analyzing `SharedUserData` for temporal addresses it should become clear that the `SystemTime` attribute is by far the most useful and potentially feasible attribute due to its scale and update period. In order to successfully leverage it in conjunction with an exploit, though, the viable opcode windows must be calculated so that a time to strike can be selected. This can be done prior to determining what the actual date is on a target machine but requires that the storage capacity (size of the temporal address in bytes), the update period, and the scale be known. In this case, the size of the `SystemTime` attribute is 12

[2]See the `InterruptTime` sub-section for a structure definition

bytes, though in reality the 3rd attribute, `High2Time`, is exactly the same as the second, `High1Time`, so all that really matters are the the first 8 bytes. Doing the math to calculate per-byte durations gives the results shown in figure 4.1. This indicates that it is only worth focusing on opcode permutations that start at byte index four due to the fact that all previous byte indexes have a duration of less than or equal to one second. By applying the scale as being measured since Jan 1, 1601, all of the possible permutations for the past, present, and future can be calculated as described in chapter 5. The results of these calculations for the `SystemTime` attribute are described in the following paragraphs.

In order to calculate the viable opcode windows it is necessary to have identified the viable set of opcodes. In this case study a total of **320** viable opcodes were used (recall that opcode in this case can mean one or more instruction). These viable opcodes were taken from the Metasploit Opcode Database[2]. After performing the necessary calculations and generating all of the permutations, a total of **3615** viable opcode windows were found between Jan. 1, 1970 and Dec. 23, 2037. Each viable opcode was broken down into groupings of similar or equivalent opcodes such that it could be made easier to visualize. Figure 7.1 shows a graph of all of the viable opcode windows between 1970 and 2038 as broken down by opcode groupings.

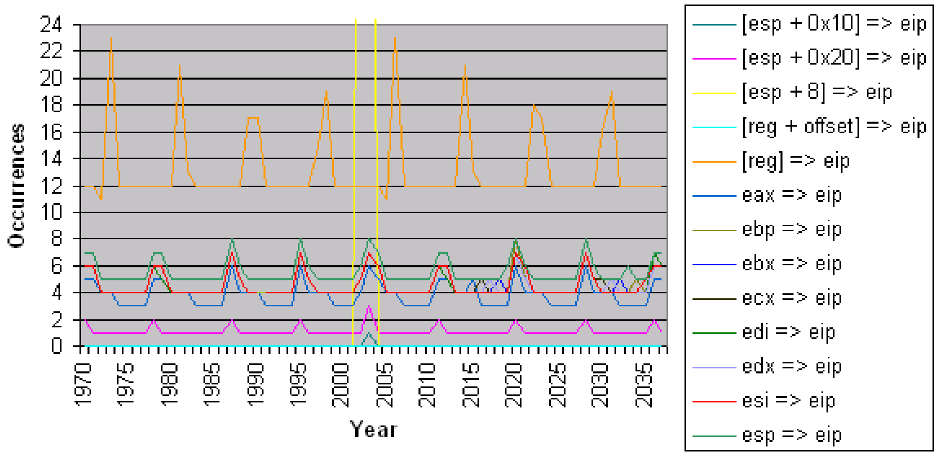


Figure 7.1: Opcode occurrences by year (size=8, period=100nsec, scale=1601epoch)

Looking closely at figure 7.1 it can bee seen that there were two large spikes around 2002 and 2003 for the `[esp + 8] => eip` opcode group which includes `pop/pop/ret` instructions common to SEH overwrites. Looking more closely at these two years shows that there were two significant periods of time during 2002 and 2003 where the stars aligned and certain exploits could have used the

SystemTime attribute as a temporal return address. Figure 7.2 shows the spikes in more detail. It's a shame that this technique was not published about during those time frames! Never again in the lifetime of anyone who reads this paper will there be such an occurrence.

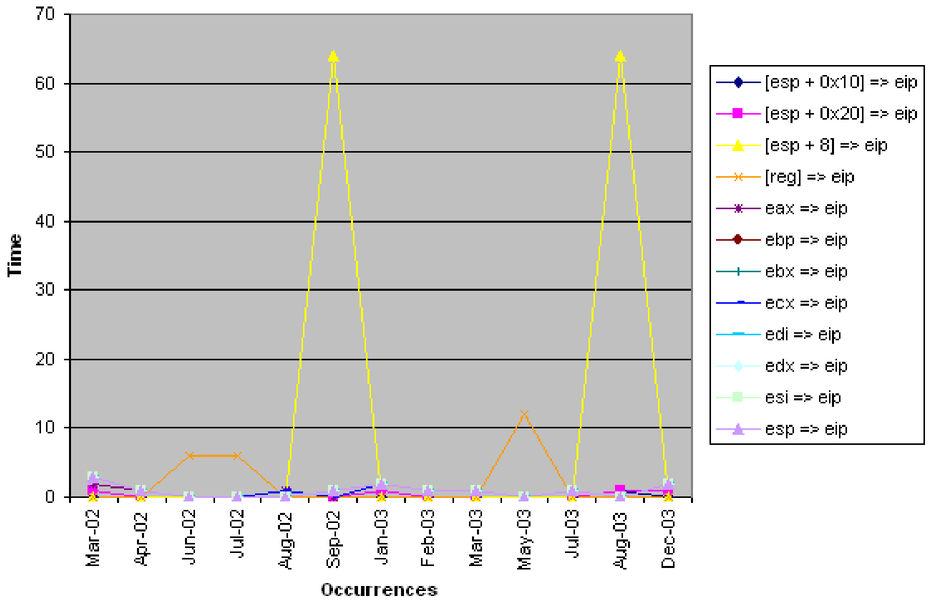


Figure 7.2: `[esp + 8] => eip` spikes in 2002, 2003

Perhaps of more interest than past occurrences of certain opcode groups is what will come in the future. The table in figure 7.3 shows the upcoming viable opcode windows for 2005.

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
Tue Sep 27 04:41:21 CDT 2005	eax => eip
Tue Sep 27 04:48:30 CDT 2005	ecx => eip
Tue Sep 27 04:55:40 CDT 2005	edx => eip
Tue Sep 27 05:02:49 CDT 2005	ebx => eip
Tue Sep 27 05:09:59 CDT 2005	esp => eip
Tue Sep 27 05:17:08 CDT 2005	ebp => eip
Tue Sep 27 05:24:18 CDT 2005	esi => eip
Tue Sep 27 05:31:27 CDT 2005	edi => eip
Tue Sep 27 06:43:02 CDT 2005	[esp + 0x20] => eip
Fri Oct 14 14:36:48 CDT 2005	eax => eip
Sat Oct 15 21:09:19 CDT 2005	ecx => eip
Mon Oct 17 03:41:50 CDT 2005	edx => eip
Tue Oct 18 10:14:22 CDT 2005	ebx => eip
Wed Oct 19 16:46:53 CDT 2005	esp => eip
Thu Oct 20 23:19:24 CDT 2005	ebp => eip
Sat Oct 22 05:51:55 CDT 2005	esi => eip
Sun Oct 23 12:24:26 CDT 2005	edi => eip
Thu Nov 03 23:17:07 CST 2005	eax => eip
Sat Nov 05 05:49:38 CST 2005	ecx => eip
Sun Nov 06 12:22:09 CST 2005	edx => eip
Mon Nov 07 18:54:40 CST 2005	ebx => eip
Wed Nov 09 01:27:11 CST 2005	esp => eip
Thu Nov 10 07:59:42 CST 2005	ebp => eip
Fri Nov 11 14:32:14 CST 2005	esi => eip
Sat Nov 12 21:04:45 CST 2005	edi => eip

Figure 7.3: Opcode windows for Sept 2005 - Jan 2006

Chapter 8

Case study: Example application

Aside from Windows' processes having `SharedUserData` present, it may also be possible, depending on the application in question, to find other temporal addresses at static locations across various operating system versions. Take for instance the following example program that simply calls `time` every second and stores it in a local variable on the stack named `t`:

```
#include <windows.h>
#include <time.h>

void main() {
    unsigned long t;

    while (1) {
        t = time(NULL);
        SleepEx(1000, TRUE);
    }
}
```

When the telescope program is run against a running instance of this example program, the results produced are:

```
C:\>telescope 3004
[*] Attaching to process 3004 (5 polling cycles)...
[*] Polling address space........
```

```
Temporal address locations:

0x0012FE24 [Size=4, Scale=Counter, Period=70 msec]
0x0012FE88 [Size=4, Scale=Counter, Period=1 sec]
0x0012FE9C [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]
```

Judging from the source code of the example application it would seem clear that the address `0x0012ff7c` coincides with the local variable `t` which is used to store the number of seconds since 1970. Indeed, the `t` variable also has an update period of one second as indicated by the telescope program. The other finds may be either inaccurate or not useful depending on the particular situation, but due to the fact that they were identified as counters instead of being relative to one of the two epoch times most likely makes them unusable.

In order to write an exploit that can leverage the temporal address `t`, it is first necessary to take the steps outlined in this document with regard to calculating the duration of each byte index and then building a list of all the viable opcode permutations. The duration of each byte index for a four byte timer with a one second period are shown in figure 8.1.

Byte Index	Seconds (ext)
0	1 (1 sec)
1	256 (4 mins 16 secs)
2	65536 (18 hours 12 mins 16 secs)
3	16777216 (194 days 4 hours 20 mins 16 secs)

Figure 8.1: 4 byte 1sec per-byte durations in seconds

The starting byte index for this temporal address is byte index one due to the fact that it has the smallest feasible window of time for an exploit to be launched (4 mins 16 secs). After identifying this starting byte index, permutations for all the viable opcodes can be generated. All the permutations from 1970 to 2038 are shown in figure 8.2.

Nearly all of the viable opcode windows conveyed in figure 8.2 have a window of 4 minutes. Only a few have a window of 18 hours. To get a better idea for what the future has in store for a timer like this one, table 8.3 shows the upcoming viable opcode windows for 2005.

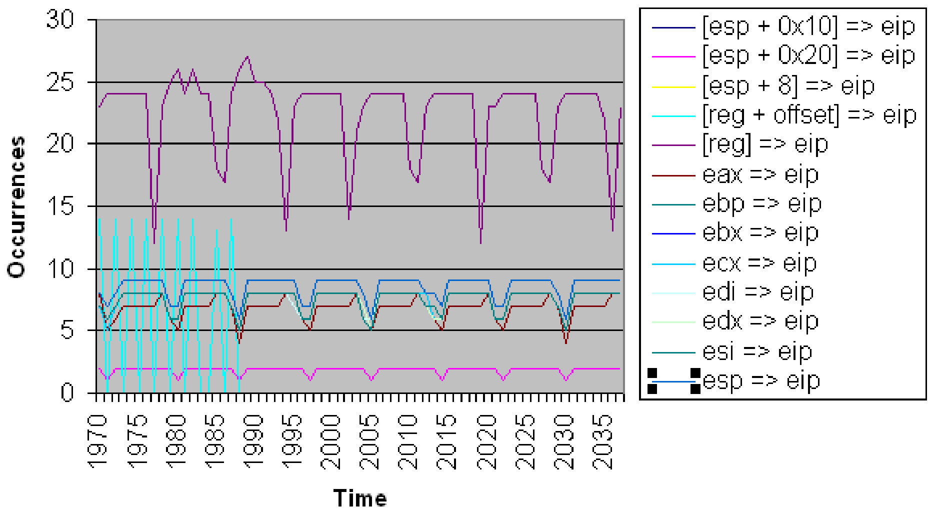


Figure 8.2: Opcode occurrences by year (size=4, period=1sec, scale=1970epoch)

Date	Opcode Group
Fri Sep 02 01:28:00 CDT 2005	[reg] => eip
Thu Sep 08 21:18:24 CDT 2005	[reg] => eip
Fri Sep 09 15:30:40 CDT 2005	[reg] => eip
Sat Sep 10 09:42:56 CDT 2005	[reg] => eip
Sun Sep 11 03:55:12 CDT 2005	[reg] => eip
Tue Sep 13 10:32:00 CDT 2005	[reg] => eip
Wed Sep 14 04:44:16 CDT 2005	[reg] => eip

Figure 8.3: Opcode windows for Sept 2005 - Jan 2006

Chapter 9

Conclusion

Temporal addresses are locations in memory that are tied to a timer of some sort, such as a variable storing the number of seconds since 1970. Like a clock, temporal addresses have an *update period*, meaning the rate at which its contents are changed. They also have an inherent *storage capacity* which limits the amount of time they can convey before being rolled back over to the start. Finally, temporal addresses will also always have a *scale* associated with them that indicates the unit of measure for the contents of a temporal address, such as whether it's simply being used as a counter or whether it's measuring the number of seconds since 1970. These three attributes together can be used to predict when certain byte combinations will occur within a temporal address.

This type of prediction is useful because it can allow an exploitation chronomancer the ability to wait until the time is right and then strike once predicted byte combinations occur in memory on a target machine. In particular, the byte combinations most useful would be ones that represent useful opcodes, or instructions, that could be used to gain control over execution flow and allow an attacker to exploit a vulnerability. Such an ability can give the added benefit of providing an attacker with universal return addresses in situations where a temporal address is found at a static location in memory across multiple operating system and application revisions.

An exploitation chronomancer is one who is capable of divining the best time to exploit something based on the alignment of certain bytes that occur naturally in a process' address space. By making use of the techniques described in this document, or perhaps ones that have yet to be described or disclosed, those who have yet to dabble in the field of chronomancy can begin to get their feet wet. Viable opcode windows will come and go, but the usefulness of temporal addresses will remain for eternity... or at least as long as computers as they are known today are around.

The fact of the matter is, though, that while the subject matter discussed in this document may have an inherent value, the likelihood of it being used for actual exploitation is slim to none due to the variance and delay between viable opcode windows for different periods and scales of temporal addresses. Or is it really that unlikely? Vlad902 suggested a scenario where an attacker could compromise an NTP server and configure it to constantly return a time that contains a useful opcode for exploitation purposes. All of the machines that synchronize with the compromised NTP server would then eventually have a predictable system time. While not completely fool proof considering it's not always known how often NTP clients will synchronize (although logs could be used), it's nonetheless an interesting approach. Regardless of feasibility, the slave that is knowledge demands to be free, and so it shall.

Bibliography

[1] Mesander, Rollo, and Zeuge. *The Client-To-Client Protocol (CTCP)*. http://www.irchelp.org/irchelp/rfc/ctcpspec.html; accessed Aug 5, 2005.

[2] Metasploit Project. *The Metasploit Opcode Database*. http://metasploit.com/users/opcode/msfopcode.cgi; accessed Aug 6, 2005.

[3] Postel, J. *RFC 792 - Internet Control Message Protocol*. http://www.ietf.org/rfc/rfc0792.txt?number=792; accessed Aug 5, 2005.