Win32k Smash the Ref
New Bug Class and Exploitation Techniques

To: Bounty for Defense @ Microsoft

Gil Dabah, 2019
distorm@gmail.com
http://ragestorm.net/blogs

Abstract

Around the year 2010 there was a spike in the number of bugs in the user interface win32 kernel (win32k) component in Windows. Most of them were due to the same bug class, caused by a lack of proper object memory management when calling back to user-mode from the kernel, which could lead to an attack of such objects. Specifically, by freeing objects from user-mode callbacks, while, back in the kernel, it would still use these non-existent objects. An unprivileged attacker could eventually forge objects that the kernel would use instead of the freed ones in order to exploit these vulnerabilities and compromise the system. This attack type is known as use-after-free (UAF).

The prior research inspired us to continue the research in this direction. Our novel bug class also attacks the same kernel objects due to a complex side-effect of destroying objects in certain locations and timing in the code, which also leads to an exploitable UAF condition. Crafting objects in such a way that when they're destroyed in the kernel, they can unexpectedly either free other objects with them, or callback to user-mode allowing the attacker to synchronize the race condition and succeed in controlling the same memory of these now-non-existent objects. This bug class is abundant in the win32k code base and this research led to finding over 25 different potential vulnerabilities, out of which 11 were exploited to prove feasibility for elevation-of-privilege (EOP). Proof of concepts (POC) are also attached for the latter, with never-seen-before techniques, and they were tested and reproduced on a guest account on latest WIP (Windows Insider Preview updated last in September 2019). This paper focuses on the multiple variants of this exploitable bug class, exploitation techniques required for the attack to work, and the pattern of finding more such vulnerabilities. This paper also suggests mitigations to defend against this type of attack in the future, hoping that Microsoft will be able to kill this bug class entirely.

Volume I

1 Introduction

The code base of win32 GUI kernel drivers (win32kfull.sys, win32kbase.sys, etc.) is very complex and there are many mechanisms and design patterns standing still that are dated from its original creation in user-mode. This paper focuses on the *UI objects reference counting mechanism* of the Handle Manager component and shows that it is vulnerable to a new type of attack that affects all versions of Windows including latest Windows 10.

A prior research in the domain was published by Garnier [1] in Uninformed volume 10 article 2, and later thoroughly presented in "Kernel Attacks through User-Mode Callbacks" by Mandt [2]. In this paper we present the "Next Generation" of win32k attacks.

1.a Win32k user-mode callback vulnerabilities in a nutshell

Since win32k was originally designed and written (in C) to run in user-mode and only later it was moved to the kernel, it must do many callbacks to user-mode to accomplish its original job. Any function that calls back to user-mode is prefixed with 'xxx' in its name to imply this *potentially dangerous* kernel to user roundtrip.

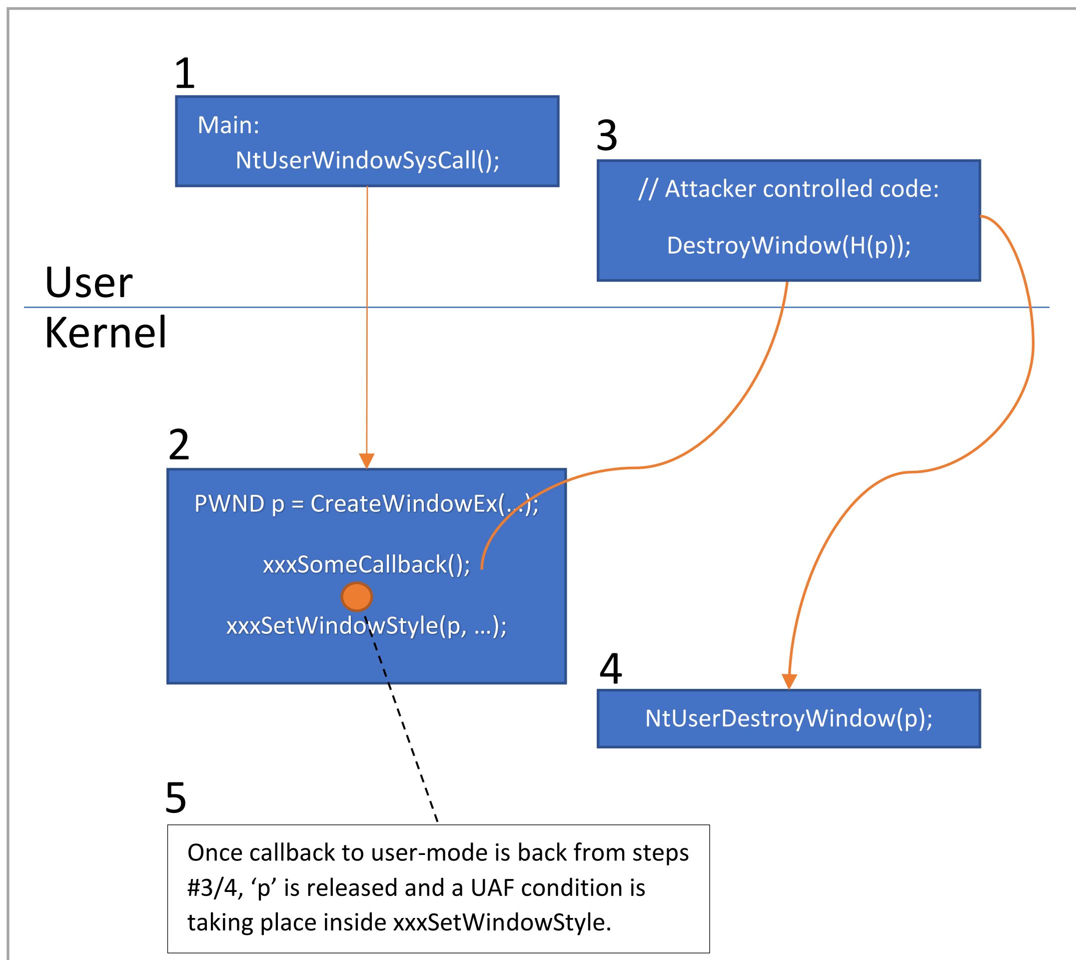
A classic known use-after-free memory attacking technique on win32k objects can occur in cases where a temporary lock (reference count increment) was forgotten on the object by the developer before a user-mode callback takes place. In such cases, the object can be destroyed and freed by an attacker from user-mode and when execution returns to the kernel and continues, the object is already freed, hence resulting in a UAF.

Reiterating the essence of the problem, there's a simple use-after-free bug in the following pseudo kernel function:

```
NtUserWindowSysCall()
{
  PWND p = CreateWindowEx(…);
  xxxSomeCallback(); // xxx calls back to user-mode to a possibly
attacker-controlled code, which can call DestroyWindow on (handle of) 'p'
in turn.

  xxxSetWindowStyle(p);
// By now 'p' points to an already freed object, therefore operating on
freed memory that an attacker can control too by allocating another object
and exploit with type confusion…
}
```

The following figure shows the exploitation flow step by step, for the above pseudo code, with the transitions from user-mode to kernel and vice versa:



1.b Mitigation to user-mode callback attacks

The fix is trivial – it's required to increment the reference count of the object and release it later, but it's a tedious task to find all instances of this bug in the code base. Today such bugs hardly exist anymore.

To mitigate the problem, it's simply necessary to temporarily lock the window object, such as:

```
PWND p = CreateWindowEx(…);
// This is a temporary lock (technically increments the refcount by 1).
ThreadLock(p);
// An attacker can still try to destroy the window here from user-mode, but it's still referenced at least once and won't be destroyed.
xxxSomeCallback();
// Zombie window is used next, but that's okay, code is mostly immune to this behavior.
xxxSetWindowStyle(p);
// p now has the last reference, given it was destroyed from user-mode. The next unlock will free the object for good, but no UAF could happen!
ThreadUnlock();
```

Unfortunately, many internal win32 kernel objects, that aren't directly exposed to user-mode, didn't use the Handle Manager's reference counting mechanism and therefore were also susceptible to UAF attacks. Such as popupmenu, menustate, window-class, and other structures. In recent years Microsoft

fixed them, sometimes even with C++ code. Microsoft used similar techniques to harden UI objects too in some places.

1.c The new attacking concept in a nutshell

Our contribution to the previous attacking technique is thanks to how object destruction works in win32k. It's possible to *link* victim objects to a target object, in such a way that once the target object is destroyed, it will destroy the victim objects as well by its own destruction.

For example, once a target window object is destroyed, it will also try to destroy its victim child windows. Although practically, it's much more complicated than that and this is what the rest of the paper explains.

The following naïve snippet is part of a real function from win32k, that later we will examine in more detail, but for now, let's see what's going on here:

```
ThreadLock(pwndParent);
pwnd = xxxCreateWindow(..., pwndParent, ...);
ThreadUnlock(); // This parent unlocking is dangerous!
if (NULL == pwnd) return ...
// From now on the child window exists and can be used.
xxxSendMessage(pwnd...);
```

Simply put, a child window is created for a parent window. That parent window is locked for that period because the window creation calls back to user-mode (so nobody can pull the classical attack here), and once it's over, it unlocks the parent window and continues to use the child window if the creation succeeded.

In our attack, we found a way to create a situation where the parent window will be gone (freed) in the ThreadUnlock function, that's done simply by exploiting the fact that the kernel calls back to user-mode in the window creation routine and we destroy the parent window from user-mode, even though the parent window is locked! Although it's still not freed, that's okay.

Once inside ThreadUnlock the parent window will be really gone, because it has 1 reference(s), now it's getting decremented to 0, and therefore its own destruction will take place, and it will also destroy the child window too. This causes a UAF condition where xxxSendMessage uses a non-existent window object that an unprivileged attacker could eventually exploit.

The following is a flow diagram that shows the attack:

Kernel

1

```
SomeKernelFunction():
ThreadLock(pwndParent);
pwnd =
   xxxCreateWindow(..., pwndParent, ...);
ThreadUnlock();
```

2

```
ThreadUnlock(): // Unlock the parent.
object = tls->lockedObjs->pop();
object->refcount -= 1; // Decrement ref!
If (0 == object->refcount)
{
  xxxDestroyWindow(object);
}
```

3

```
xxxDestroyWindow(object):
...
if (NULL != object->childWnd)
{
   xxxDestroyWindow(object->childWnd);
  // pwnd of SomeKernelFunction
  // is now freed too!
}
...
HMFreeObject(object); // Free memory!
```

4

```
SomeKernelFunction(): // Continues
if (NULL == pwnd) return ...
xxxSendMessage(pwnd...); // UAF!
```

This diagram shows how a memory use-after-free attack works on the created child window from the death of the parent window. Given that the parent window was destroyed from the callback to user-mode from xxxCreateWindow. It gets back to kernel with a single reference left, and ThreadUnlock decrements the last reference, which leads to its destruction, and by a chain-effect it destroys its child window too, which wasn't locked yet.

For brevity the above diagram is not fully accurate but conveys the attacking concept, which is generic and applies to different types of objects as we shall see in the rest of the paper. In section 3.b we show exactly how the attack is done for this example.

Unlike the prior research attacking technique - since there's no callback to user-mode at the time the object is really freed, in order to be able to forge a replacement fake window object for the exploit to succeed, there's a race condition between the time the window object was freed and the time it's used once again and that challenge is discussed and solved later too.

1.d Outline

The next section (#2) gives the reader the initial technical knowledge on how UI objects are managed in win32k. The section afterwards (#3) is the main one describing in high detail the novel attacking technique with various exploitation methods.

The second volume of this paper is about escaping back to user-mode from totally unexpected locations in the code that can even lead to more vulnerabilities and winning the race condition mentioned earlier for good. It also covers the vulnerabilities found throughout this research and suggests many mitigations to close all variants for the future too.

2 Background

The Handle Manager is the component that helps to manage the lifetime of UI kernel objects. It uses reference counting and an additional state machine to determine the status of objects. The Handle

Manager allocates the objects themselves and corresponding unique ‘handles’ to let user-mode applications access these objects. Likewise, the Handle Manager is responsible to free the object and the object-handle under certain conditions.
The objects are exposed to user-mode by means of APIs (eventually syscalls to kernel) and handles. In the kernel things get much more complicated.

2.a UI objects reference counting

UI objects [3] in win32 kernel are *manually* reference counted. Unlike ‘smart pointers’ in OOP languages, ‘manually’ means that the developer must keep track of the locks and unlocks in order and place inside the code, which is error prone.

These objects have a Create and a corresponding Destroy APIs. In the kernel, using the Handle Manager, the constructor function allocates the object and a handle.

Upon creation it **initializes** the one and only **reference count to zero**. This helps to explain the problem in the above *classical* UAF example, and why locking was missing.

UI objects lock types

There are only two types of locks that objects can use: a temporary lock and a permanent lock. Both lock-types similarly increment and decrement the same reference count once they lock and unlock correspondingly.

A temporary lock (ThreadLock function) is like a short-lived scope lock and it’s used when the kernel calls back to user-mode, and once control is returned, it unlocks (ThreadUnlock) the object. Thus, blocking user-mode from releasing an in-use object.

A permanent lock is used when one object is linked to another, like when a menu has a sub-menu.
A helper function (HMAssignmentLock) initializes a dumb C pointer with the target object’s address, and most importantly, increments the target object’s reference count (refcount, in short). Once this link is ended, whose timing depends on the design of the specific UI object and its APIs, the pointer gets nulled and the target object’s refcount is decreased (by HMAssignmentUnlock).

UI objects destruction

The **destroy function checks for zero references** and only then it **really destroys and frees** the object and the handle.

If the destroy function is called on an object with a positive number of references - the destruction behavior depends on the type of the object, but basically the following is true:
The destroy function only marks the object for **future-destruction**. And as soon as the refcount drops to zero, by *either* lock type, wherever that happens, the object will get destroyed right then and there!

If an object’s refcount is back to 0, without ever being destroyed, the object will continue to exist normally.

2.b Zombie objects

There are three states objects can have:

1) An object has zero references
2) An object has positive number of references; locked by other objects or temporarily.

3) And a **destroyed** object with positive references, which is still allocated and *semi-alive*.

The last state, semi-alive object, is unofficially called a 'zombie' object by Microsoft employees [4], some might call it a 'corpse', we will stick to zombies throughout this paper as we think it's more appropriate. A zombie window, for instance, is a window that is still referenced by other objects or temporarily, which was already destroyed by DestroyWindow, but cannot free its memory and handle just yet as the kernel still uses it.

A zombie object will be really gone once all references to it are decremented and the refcount is back to 0, but until then, user-mode can't touch the object anymore. It will be really destroyed precisely at the same location (calling site) where the last Unlock of any variation happen.

Upon a syscall entrance a handle validation takes place for relevant arguments, rendering zombie objects-handles inaccessible to user-mode. For example, the syscall code calls ValidateHmenu or ValidateHwnd on the untrusted input handle arguments from user-mode, if the return value is NULL, it will immediately return to user-mode with an error value, thus blocking access to zombie objects as if they do not exist, even though they truly do.

Mitigation for zombie objects access with pseudo code:

```
BOOL NtUserDestroyWindow(HWND hwnd)
{
  PWND pwnd = ValidateHwnd(hwnd);
  if (NULL == pwnd)
  {
    SetLastError(error-invalid-handle);
    return 0;
  }
  return xxxDestroyWindow(pwnd);
}
```

This also explains why it's only possible to call DestroyWindow API on an object once.

Nevertheless, the window destruction behavior is such that it unlinks owned objects (sub-resources) *immediatel*y upon the *first* call to DestroyWindow, **even if the refcount is positive**. Sub-resources such as child windows, caret, mouse capture state, menu, icon, timers, and many others, if any exist at all. Now, depending on their own refcount, they might get destroyed and freed too, i.e. the **objects chain-effect**.

2.c Window destruction behavior

In this section we will see how xxxDestroyWindow treats its sub-resources and how it decides eventually to free itself. The following is a **very** rough pseudo code of xxxDestroyWindow mixed with xxxFreeWindow:

```
void xxxDestroyWindow(PWND pwnd)
{
  …
  xxxFW_DestroyAllChildren(); // Destroy child windows, if exist!
  …
  if (NULL != pwnd->spmenu) // If there’s a menu, remove and destroy it.
  {
    PMENU tmp = pwnd->spmenu;
    if (HMAssignmentUnlock(&pwnd->spmenu)) // If it’s still locked -
    {
      DestroyMenu(tmp); // Try destroying it (it can remain a zombie).
    }
  }
  …
  if (pwnd == ptiCurrent->pq->capturedPwnd)
    xxxReleaseCapture();
  …
  DereferenceClass(pwnd);
  …
  if (HMMarkObjectDestroy(pwnd)) // Check for zero refs!
    HmFreeObject(pwnd); // Only now free the object and handle pair.
}
```

The point is that some objects are *owned* by the window and other objects are just *borrowed*, it depends on the implementation of the window and other UI objects. Meaning that they are all referenced by the window, but the difference is whether the window will try to destroy them upon unlocking them (like the menu object in the snippet). And anyway, if a sub-resource object is destroyed from user-mode and gets the last reference decremented by the window’s destroy function, it gets freed too, which creates the said chain-effect.

3 Abusing the last reference

This new technique revolves around the last reference of UI kernel objects, and it creates a special memory UAF opportunity, that can eventually lead to an elevation of privileges from guest to kernel.

The bug itself is not in the last reference or its destruction code per se, rather it's a situation with a mixture of various hard constraints. It happens in the call site to the unlocking code of the last reference where the side effect of taking down *additional* linked objects results in a potentially hidden collateral damage, i.e the mentioned chain-effect.

DestroyWindow, for example, releases sub-resources on the first call. And afterwards once the window is no longer referenced, it gets fully destroyed *once again* by the Handle Manager by design. Normally, nothing much happens in the second time, as all sub-resources were already released in the first time…

However, it’s possible to change this, by making a zombie object trigger unexpected chain-effect in its final destruction.

The elegant trick is that under certain circumstances it is still possible to **make some changes** to a zombie object even after it was already destroyed once and is inaccessible to user-mode any longer. From now on we will refer to this method as Zombie Reloading (with chain-effect, more below).

Therefore, once an object is really destroyed for the last - technically second - time, it can wreak havoc due to this *unexpected behavior*!

The pattern to trigger this bug class requires some must-have conditions for successful exploitation. We will refer to such call sites as **smashable sites**:
(Note ‘decref’ is object’s reference count decrement, e.g. unlock)

1. A dumb pointer that’s assigned to a **victim** UI **object** (no locking involved).
2. A **target** window object’s **last decref** functionality (such as, ThreadUnlock, HMAssignmentUnlock, or HMAssignmentLock – which may decref a previously held object!).
3. Control where the target’s last decref happens.
4. Zombie reload **target** object to **link*** the **victim** object(s) that will get freed.
5. Use of **victim object** after steps #2 and #3.

*Linking means that either directly or indirectly it will eventually cause a memory free.

To summarize, a zombie object is an object that was destroyed but still has a positive number of references. Once the object refcount reaches 0, the Handle Manager will free the object after invoking its destruction functionality. Therefore, if a *victim* sub-resource can be linked to a *target* zombie object *without explicitly incrementing the parent’s refcount*, it will get destroyed together with its parent. Thus, leading to a UAF condition on the sub-resource. Given that the victim objects aren’t locked, or getting their last reference unlocked in this chain-effect.

We call the zombie object a **target** as well, because that’s the object we’re concentrating on mostly in the attack and research, where the magic happens in the unlocking, and we reload it with the **victim** object(s) that we actually apply, so to speak, the futuristic UAF on. In some attacks that are a few zombies involved as we will see in the advanced technique.

Note that it’s impossible to satisfy the second condition of the above pattern as long as invoking syscalls that take objects (handles) as arguments as those are temporarily locked for the entire syscall duration until they return to usermode.

There are a few methods to get this victim sub-resource attacked, but first we need to attach the sub-resource to the target zombie object.

3.a Zombie reloading technique

In this section we will see how to reload a zombie. Technically, reloading means that we modify a zombie object although it was already destroyed previously. A modification can be setting some attribute of the object, or linking it to another object, or many other possible manipulations that will aid eventually in attacking a target smashable site. The reloading operation is not the end goal, but rather it’s a by-product of the required chain-effect that causes the actual attack.

A couple of important obstacles we will need to bypass:

1) Win32k syscalls always validate their input upon entrance (or later in certain cases), and if an object was marked as destroyed but still exists (i.e. a zombie window), then the validation will fail.
   For instance, if we call SetTimer API from user-mode on a zombie, the operation will fail with invalid-handle status.
2) The first call to DestroyWindow will unlink its sub-resources too. Meaning that the final destruction won't unlink any sub-resources, because none exist anymore. No sub-resources linked - no bug.

Therefore, we need to come up with a technique to modify a zombie object bypassing the above constraints. If we manage to do it, and we can satisfy condition #4 (linking by zombie reload) of a smashable site, we then got a working attack.

Notice how the pattern in the next examples is the same: the zombie object is manipulated in kernel after there's a callback to user-mode where the window is destroyed, thus once execution continues inside the kernel, it operates on a zombie window. And in most cases the kernel won't check if the object is destroyed.

Imagine the following code inside the kernel, assuming we need a timer for a zombie (later we will see such a scenario):

```
ThreadLock(pwnd);
xxxSomeCallback(); // Here we can destroy pwnd from user-mode.
// Making this window a zombie now.
// Lo and behold what happens next!
InternalSetTimer(pwnd, ...);
ThreadUnlock();
```

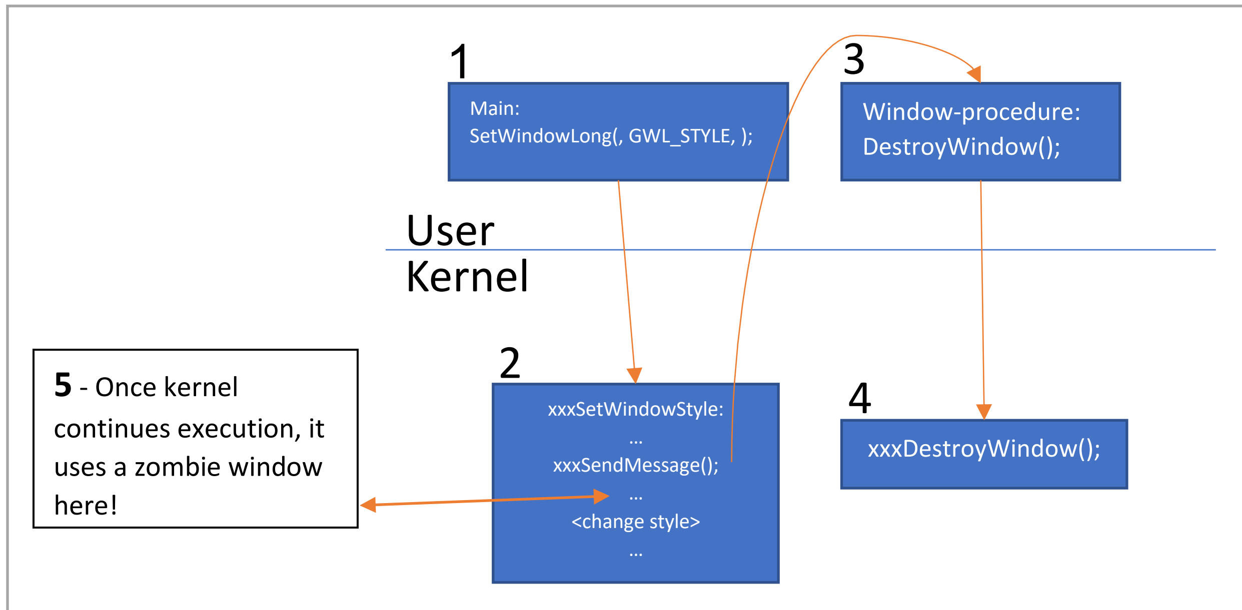
Basically, nothing stops the kernel itself from setting a timer for the zombie window. We just need to find a function that eventually behaves this way (e.g. xxxFlashWindow). It's true that we don't control the arguments that are passed to InternalSetTimer, but that's fine as it's not always necessary.

Another example in kernel equivalent of SetWindowLong for changing window-style (GWL_STYLE) API:

```
LONG xxxSetWindowStyle(pwnd, newStyle)
{
  LONG oldStyle = pwnd->style;
  xxxSendMessage(pwnd, WM_STYLECHANGING, …);
  …
  pwnd->style = newStyle;
  // Zombie got a new style.
  …
  return oldStyle;
}
```

Once the window is destroyed in user-mode while handling the WM_STYLECHANGING message, it returns to kernel and continues execution, this time 'pwnd' points to a zombie window, but changing its style whatsoever, thus 'reloading' it.

This is a flow diagram of zombie reloading technique for SetWindowLong:



Yet another useful example of a zombie manipulation is to attach an owner window (SetWindowLong API with GWL_HWNDPARENT index). Here's a very rough pseudo code:

```
LONG xxxSetWindowLong(pwnd, index, newValue)
{
  LONG oldValue = 0;
  …
  if (index == GWL_HWNDPARENT)
  {
    PWND newOwner = ValidateHwnd((HWND)newValue);
    if (NULL == newOwner) return error-invalid-handle;
    oldValue = pwnd->ownerWnd->HWND;
    xxxHandleOwnerSwitch(&pwnd->ownerWnd, newOwner);
  }
  else if (index == …
  …
  return oldValue;
}
```

One might argue how come internal kernel functions operate on zombie windows is legit, but that's part of the design - the kernel can't really release the window until all other functions in the kernel are done using it. What wasn't part of the design is the fact that we use it against itself.

The generic reloading pattern, for UI objects, consists of:

1) A callback to user-mode. Any of:
   a. Window messaging procedure
   b. Hook procedure
   c. Client helper procedure
2) From user-mode - destroy the object to make it a zombie.

3) Back in kernel – **flow continues to modify** the zombie object.

Most of the adventure of researching a potential decref site is to see if it's possible to turn it into a smashable site. Or in other words, to manage to unlock the target object from all permanent and temporary locks, such that the last decref happen in a certain controlled location and take down a victim object with it.

This means that if we see a potential decref site with a target object that we try to smash, we will need to find first a way to manipulate it in such a way that a chain-effect (attaching more victim objects, for instance) can happen with that zombie object.

Sometimes it's not possible to reload a zombie with a needed chain-effect and it renders a smashable site immune to this attack. Practically some internal kernel functions check for a destroyed window state and fail, but it happens because it is erroneous by design in certain places (e.g. kernel xxxSetParentWorker will exit if either of the parent/child windows are destroyed in the middle of its action).

3.b Attacking the kernel with zombie object chain-effect

Now that we know how the new attack works and how to reload a zombie, we can revisit our first example. The following snippet is a paraphrase based on the vulnerable xxxMnOpenHierarchy:

```
Line 1: ThreadLock(pwndParent);
Line 2: pwnd = xxxCreateWindow(..., pwndParent, ...);
Line 3: ThreadUnlock(); // This parent unlocking is dangerous!
Line 4: if (NULL == pwnd) return ...
Line 5: // From now on the child window exists and can be used.
Line 6: xxxSendMessage(pwnd...);
```

Before we dive into the details of this attack, it's also important to say that some might think it's possible to destroy the child window while it's created, as well as the parent, from the callbacks to user-mode inside xxxCreateWindow, and then somehow you get a UAF at line #6. In reality, xxxCreateWindow will return NULL and line #4 will exit the function. Hence, to cause an actual UAF at line #6, it requires a smarter attack, with zombie reloading, etc.

According to our pattern, we show that each condition is satisfied:

1. A dumb pointer that's assigned to a **victim** UI **object** – see line #2.
2. A **target** window object's **last decref** functionality – see line #3.
3. Control where the target's last decref happens – yes, we can remove other locks.
4. Zombie reload **target** object to **link** the **victim** object(s) that will get freed – yes.
5. Use of **victim object** after steps #2 and #3 – see line #6.

Exploiting this specific issue practically requires a lot of research and understanding of the flows and state machines of the menu code itself, but it was eventually possible to remove the permanent locks on the parent window (using WM_NEXTMENU) and to make line #3 decrementing the last reference, for brevity we will ignore the menu internals here. (See the POC of xxxMnOpenHierarchy for more information.)

The goal is to show how smashing the last reference of the parent window can lead to a UAF situation with the child.

Unlike the classical UAF explained in the introduction, this attack doesn't leave kernel to call back to user-mode, and it makes exploitation harder as we will see later, but the UAF happens all the same.

This is the flow under the hood in order to trigger above code for a UAF:

1. Inside xxxCreateWindow at line #2 after both the parent and the newly created child window are both temporarily locked, user-mode is called back:
   1.1 Both windows are destroyed in user-mode
   1.2 At the "same time" (through a callback to user-mode) as they're destroyed, we link the parent and child windows as owner-ownee relationship through SetWindowLong GWL_HWNDPARENT (see example 'xxxSetWindowLong' above or relevant POC). This is the zombie reloading part of this attack.
2. We destroy the child inside its own xxxCreateWindow, though as it still has a temporary lock and by the parent, it's now a zombie window itself, and a valid pointer is returned, nevertheless. LOL!
   We hook the WM_CREATE message by SetWindowsHookEx (WH_CALLWNDPROC) to do the trick (the timing of self-destruction) which is very late in the window creation process and once it's done at that step, it's already too late for xxxCreateWindow to return NULL.
3. We do some juggling to release the parent window from all *permanent* references, so last reference is held by ThreadLock at line #1 shown in the snippet.
4. ThreadUnlock at line #3 releases the parent (refcount goes from 1 to 0), since we made sure it has zero references now, it calls xxxDestroyWindow, which sees that it is linked to an owner window and unlinks it too.
5. The parent then takes down the child window too - technically speaking it unlocks an owner window, and owner's reference goes to zero too, and therefore xxxDestroyWindow is invoked again to destroy the child.
6. Finally, the ThreadUnlock is over and both objects are destroyed and freed for good.
7. Profit - UAF at line #6.

Given that xxxCreateWindow successfully created a child window and returned a valid pointer; inside ThreadUnlock we managed to cause the release of the parent window and subsequently release (destroy and free) this child window too. Leading to a UAF situation for the 'pwnd' pointer in xxxSendMessage and afterwards.

This is the stack trace of how the child window is destroyed by the parent window inside the unlocking:

```
00 ffffbd89`de2de7d8 ffff819a`1d0a1745 win32kfull!xxxDestroyWindow
01 ffffbd89`de2de7e0 ffff819a`1d02d6a9 win32kbase!xxxDestroyWindowIfSupported+0x25
02 ffffbd89`de2de810 ffff819a`1d03300f win32kbase!HMDestroyUnlockedObject+0x69
03 ffffbd89`de2de840 ffff819a`1d032fad win32kbase!HMUnlockObjectInternal+0x4f
04 ffffbd89`de2de870 ffff819a`1d6570be win32kbase!HMAssignmentUnlock+0x2d
05 ffffbd89`de2de8a0 ffff819a`1d66a5b7 win32kfull!xxxFreeWindow+0x6ea
06 ffffbd89`de2de9c0 ffff819a`1d0a1745 win32kfull!xxxDestroyWindow+0x377
07 ffffbd89`de2deac0 ffff819a`1d02d6a9 win32kbase!xxxDestroyWindowIfSupported+0x25
08 ffffbd89`de2deaf0 ffff819a`1d0369a5 win32kbase!HMDestroyUnlockedObject+0x69
09 ffffbd89`de2deb20 ffff819a`1d7f1d7a win32kbase!ThreadUnlock1+0x95
0a ffffbd89`de2deb50 ffff819a`1d7f156c win32kfull!xxxMNOpenHierarchy+0x402
0b ffffbd89`de2dee10 ffff819a`1d7f0e80 win32kfull!xxxMNKeyDown+0xa6c
0c ffffbd89`de2def90 ffff819a`1d7f4654 win32kfull!xxxMNKeyDown+0x380
0d ffffbd89`de2df110 ffff819a`1d5ffadd win32kfull!xxxMenuWindowProc+0xe74
0e ffffbd89`de2df300 ffff819a`1d5ff70c win32kfull!xxxSendTransformableMessageTimeout+0x3bc
0f ffffbd89`de2df460 ffff819a`1d7ec8ed win32kfull!xxxSendMessage+0x2c
10 ffffbd89`de2df4c0 ffff819a`1d7ed18d win32kfull!xxxHandleMenuMessages+0x561
11 ffffbd89`de2df590 ffff819a`1d81471c win32kfull!xxxMNLoop+0x3d9
12 ffffbd89`de2df650 ffff819a`1d7819c7 win32kfull!xxxMNKeyFilter+0x194
```

Chain-effect: ThreadUnlock kills parent window first and then recursively kills the child window (also last ref'ed).

```
13 ffffbd89`de2df680 ffff819a`1d6183a2 win32kfull!xxxSysCommand+0xa4d23
14 ffffbd89`de2df7a0 ffff819a`1d617a50 win32kfull!xxxRealDefWindowProc+0x836
15 ffffbd89`de2df960 ffff819a`1d6c3b9c win32kfull!xxxWrapRealDefWindowProc+0x60
16 ffffbd89`de2df9d0 ffff819a`1d6b7a41 win32kfull!NtUserfnDWORD+0x2c
17 ffffbd89`de2dfa10 fffff800`06278885 win32kfull!NtUserMessageCall+0x101
```

The above pseudo code seems to fulfill the attacking conditions, and hence buggy and exploitable. It's vital to note that xxxDestroyWindow won't callback to user-mode on its final destruction by design. But that behavior won't stop us, as it's not necessary to leave to user-mode, as we've just seen. We will talk later about how to exploit it under the same circumstances.

3.c Advanced chain-effect mass destruction

At other times it's required to reload the zombie with *several* changes for a chain-effect to work effectively at the target smashable site.

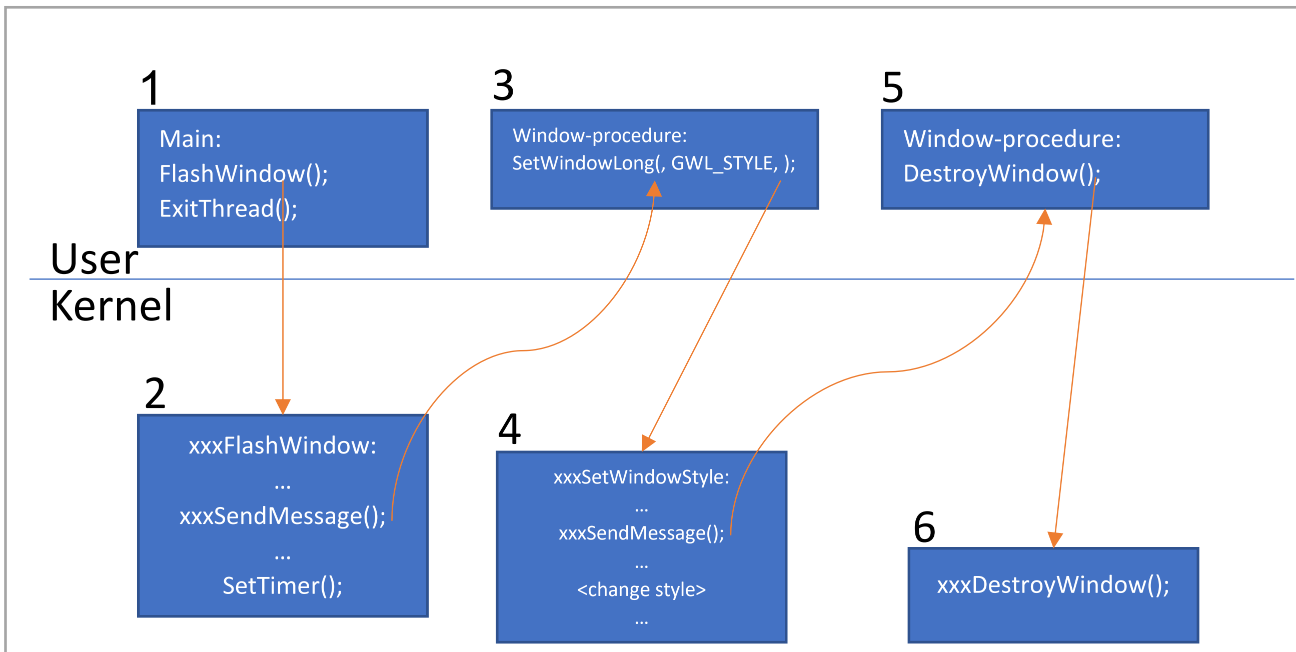
In the following timers showcase we attack the cleanup functionality, which works like this:

```
void DestroyThreadsTimers()
{
  p = g_timers_list;
  while (NULL != p)
  {
    nextTimer = p->next;
    if (timer-belongs-to-calling-thread)
    {
      FreeTimer(p);
    }
    p = nextTimer;
  }
}
```

In order to exploit the above code, we need to free two consecutive timers. The first timer will be released by Freetimer, and as a response to a chain-effect it will release an additional timer. Once the next iteration begins 'nextTimer' points to a freed memory, which is the end goal. The method to attack this code is to know that a window is associated with a timer. What we will need to do is to create a situation where there's a timer that points to a zombie window that points to another timer, e.g. timer -> window -> timer #2, which gets all freed in a chain-effect inside the call to FreeTimer cleanup function.

A timer object doesn't have to be destroyed first in order to be freed by a window, it automatically gets destroyed once it's unlinked from its associated window (again, each object has a different design).

In order to create the said chain, we will need to stack a few zombie reloadings together. This is possible due to the nature of win32k callbacks. Imagine literally stacking a few calls to user-mode, like in the following real scenario:



The flow is divided to two sections, from first to last call and backwards (collapsing), described step by step:

1) From user-mode 'main' calls to FlashWindow.
2) Inside xxxFlashWindow we go back to user-mode by window-messaging.
3) Call to SetWindowLong on same window.
4) Inside xxxSetWindowStyle we go back once again to user-mode by window-messaging.
5) At last we can destroy the window and zombify it.
6) xxxDestroyWindow will reset all styles, release sub-resources and undo other states, but we're agnostic.

And see how the stacking and user to kernel transitions create a reversing effect and reload the zombie.

7) xxxDestroyWindow is back to user-mode to #5.
8) The code returns to xxxSetWindowStyle (#4), and only now it changes the window style on a zombie – **first timer is created by composite functionality!**
9) xxxSetWindowStyle is back to user-mode to #3.
10) The code returns to xxxFlashWindow (#2), and only now **it creates a second timer associated with the zombie** window.
11) xxxFlashWindow is back to user-mode (#1) and we're done.
12) At last, it calls ExitThread to trigger DestroyThreadsTimers.

Notice how we call the APIs in a reversed order and lastly destroy the window, and from there collapse back to callers, which manipulate the zombie after it was already destroyed in the order really needed.

Finally, we got a zombie window that is composited (WS_EX_COMPOSITED) and has a timer (from FlashWindow). The reason the window must be composited is because it creates a second (global) timer, so no more references exist to the same zombie window (otherwise, it won't be the last decref). The goal was to find a way to reload a zombie with two timers (or affect/destroy somehow a second timer indirectly. The composite timer doesn't belong to the zombie window directly, but once the

window is not composited anymore upon final destruction, the timer is removed too), and this is what the research produced – first you look for a potential smashable object and then you work the exploit backwards…
Actually we don't even care about what a composited window does, we just need its timer, in this case.

This is how the chain-effect takes place inside DestroyThreadstimers in parallel to the stack trace below:

1. The zombie window still exists at step #11 as the timer holds a reference to it.
2. The reference to this timer is decremented in Freetimer as the DestroyThreadsTimers snippet shows.
3. FreeTimer unlocks the zombie.
4. Zombie window is finally destroyed and resets its composite style through SetVisible.
5. SetVisible calls DecrementCompositeCount which removes its global timer as it sees there are no other composite windows in the system (a legitimate assumption).
6. Composite's timer is removed by FindTimer which has a Boolean to indicate it should free the given timer.

Here's a stack trace of the vulnerability:
(FreeTimer is doing a single time recursion to free the next timer, and then UAFing…)

```
00 ffffde8a`b9f7a180 fffff26c`d2eade54 win32kfull!FreeTimer+0x9f
01 ffffde8a`b9f7a1b0 fffff26c`d2fd7c96 win32kfull!FindTimer+0xa4
02 ffffde8a`b9f7a200 fffff26c`d2f7873a win32kfull!DecrementCompositedCount+0x3a
03 ffffde8a`b9f7a240 fffff26c`d2f7b5ea win32kfull!SetVisible+0xf4b7a
04 ffffde8a`b9f7a270 fffff26c`d320cd85 win32kfull!xxxDestroyWindow+0xec38a
05 ffffde8a`b9f7a370 fffff26c`d31b7739 win32kbase!xxxDestroyWindowIfSupported+0x25
06 ffffde8a`b9f7a3a0 fffff26c`d31b703f win32kbase!HMDestroyUnlockedObject+0x69
07 ffffde8a`b9f7a3d0 fffff26c`d31b6fdd win32kbase!HMUnlockObjectInternal+0x4f
08 ffffde8a`b9f7a400 fffff26c`d2eadf57 win32kbase!HMAssignmentUnlock+0x2d
09 ffffde8a`b9f7a430 fffff26c`d2eadda8 win32kfull!FreeTimer+0x97
0a ffffde8a`b9f7a460 fffff26c`d31b6393 win32kfull!DestroyThreadsTimers+0x48
0b ffffde8a`b9f7a490 fffff26c`d31bc5b0 win32kbase!xxxDestroyThreadInfo+0x52b
0c ffffde8a`b9f7a630 fffff26c`d2ee22b1 win32kbase!UserThreadCallout+0x290
0d ffffde8a`b9f7a6d0 fffff26c`d3247fab win32kfull!W32pThreadCallout+0x61
0e ffffde8a`b9f7a700 fffff26c`d3ba103b win32kbase!W32CalloutDispatch+0x3db
0f ffffde8a`b9f7a930 fffff802`62531371 win32k!W32CalloutDispatchThunk+0xb
10 ffffde8a`b9f7a960 fffff802`62472d57 nt!ExCallCallBack+0x3d
11 ffffde8a`b9f7a990 fffff802`6249a2ab nt!PspExitThread+0x497
```

3.d Alternative zombie reloading

Apparently, zombie reloading doesn't have to happen like the demonstrated pattern above, and there's another type of attack vector worth mentioning because it enabled a few more vulnerabilities.

The outcome effect is all the same – we need to get a zombie window with a last reference by another object. Earlier, we showed how to do it while destroying the object in the middle of some kernel function. This time it's simpler because DestroyWindow isn't in our way.

One alternative is done by AttachThreadInput [5], which is fully documented by MSDN. The details of this function currently don't matter, but only the fact that there's a side effect of how memory management is done by the behavior of this function and we're going to abuse it. Note that it's implemented by using a thread-queue (that's PQ below), which is responsible for actual user input.

Here's a pseudo code of the kernel equivalent CreateCaret API:

```
BOOL xxxCreateCaret(pwnd, …)
{
  PQ pq = W32GetCurrentThread()->pq;
  if (NULL != pq->caret->spwnd)
  {
    zzzInternalDestroyCaret(); // Here we can free pq!
  }
  …
  LockObjectAssignment(&pq->caret->spwnd, pwnd); // UAF!
  …
}
```

It creates a new caret for the given window, however, if there's a previous caret, it will get destroyed first. As we already mentioned and can be seen in the pseudo code, a caret is associated with a window. Suppose we manage to smash a zombie window inside zzzInternalDestroyCaret, so xxxDestroyWindow is triggered eventually, it will reverse the effect of AttachThreadInput under certain conditions.

Thus, given how AttachThreadInput is detaching, **it will re-allocate some thread-queue** (PQ) structures. Making the thread's older 'pq' pointer stale and therefore UAFing.

This example shows why it's an alternative-reloading as it's not required to DestroyWindow() to zombify the window from within some user-mode callback like we did previously. Since DestroyWindow functionality cleans its own caret very early in the flow, before it's marked for destruction and the window is still accessible by user-mode; therefore, it's still possible to re-CreateCaret() **afterwards** (for example, in the WM_NCDESTROY message handling) and it becomes the only object that points to that now-zombie window. Once a new caret is created in the same thread, this zombie is freed, causing the cascade of thread-queue-detaching, making LockObjectAssignment using a freed 'pq'…

Another example is the LockWindowUpdate API which can permanently lock a window object, and even if the window is destroyed that reference isn't released (how convenient). Thus, automatically it turns the window into a zombie that has one last reference held by some kernel structure that we attack in the FreeSPB POC.

The bottom line is that assuming we need to get a situation where a zombie object is still referenced after it was firstly destroyed (recall the first destroy cleans up all sub-resources, etc), then there are more ways to do it, like shown in this section:

- By calling APIs that don't have to be called from some callback from the destruction functionality. E.g AttachThreadInput or LockWindowUpdate, and there might be more. In other words, an API that can affect some state that xxxDestroyWindow will need to revert. And this API can be called *asynchronously* from DestroyWindow.
- By creating a victim sub-resource post its cleanup phase in the process of destruction of the corresponding parent object - meaning it won't be cleaned again from this point on - so the effect will stay for the zombie, thus it's now reloaded, e.g. the caret situation.

3.e Zombie objects binary UAF exploitation

This paper presented a technique that creates a standard memory use-after-free vulnerability which is harder to exploit, because it's not done directly via user-mode callbacks like in the old fashion. And since

the thread is not leaving to user-mode, it means there is a very short time window between the time the object is freed and used again. This is in essence a classic kernel *race condition*.

Believe it or not, but all NtUser syscalls use the same critical section exclusively or shared-wise to synchronize all data structures in the kernel. This “critical section” is really an ERESOURCE kernel synchronization object but treated as such for the sake of simplicity.

This means that once a thread acquired the critical section all other thread cannot execute NtUser-syscalls code until it is done. Thus, winning the race is not possible using NtUser syscalls.

Therefore, it was necessary to find another mechanism in the kernel that can allocate the same freed object’s address which is going to be used-after-freed.

Fortunately, this can be done using GDI syscalls that share the same memory heap pool NtUser uses. As both subsystems don’t share the same ERESOURCE that NtUser syscalls use, they are not blocked from continuing to run their own logics simultaneously. Meaning it’s possible to use GDI syscalls to try to win the race.

However, if the race is lost, there won’t necessarily be a BSOD as kernel memory management leaves memory space of the object as is (unless, there’s a mitigation and we will discuss it later). Remember also that we talk about an exclusive memory pool that is accessible only by win32k, so less chances for other kernel components to interfere with it.

Experimenting with winning the race, before kernel LFH (Low-Fragmentation-Heap) was introduced to the kernel of Windows 10, proved to be easy with just another thread. However, no testing was done with kLFH in place. Although kLFH is not immune to such attack, and it will probably only require more heap shaping games like spraying similar sized objects…

To successfully exploit these vulnerabilities, we have to win a hard race condition. Additionally, it's impossible in some cases to link a victim object with a target zombie object, leaving us with no usable chain-effects. Therefore, we continued seeking for a better attack.

Volume II

4 The ultimate zombie reloading

Suppose there's a way to call back to user-mode **from the last zombie-window unlocking site**, or really from xxxDestroyWindow once again, even though the window was already destroyed. Remember, final destruction must never call back to user-mode.

But if it's possible it will be a game-changer, no race conditions needed to be won, no potential blue screens, just attack the kernel through user-mode callbacks like it was possible before object-locking-awareness in a synchronized fashion. Then we can potentially go back to user-mode from a smashable site, and from user-mode do the destruction of any target objects as we see fit and try to catch the same freed block with infinite number of trials, then return to kernel for allegedly 100% successful exploitation. Also having the ability to attack and exploit new smashable sites that weren't possible before, because we don't need a chain-effect anymore. And we would be able to spare the need of complex zombie reloading chains.

4.a Obstacles upon reaching the (im)possible - calling back to user-mode

So far, we know that on first call to DestroyWindow API (eventually xxxDestroyWindow in kernel), naturally it calls back to user-mode to notify for object destruction and other messages (like WM_DESTROY). On the final call to xxxDestroyWindow, as was already described, no callbacks are done!

Our goal is to gain the ability to call back to user-mode from inside a **zombie window's** xxxDestroyWindow or its calls hierarchy (e.g. a function that is called by xxxDestroyWindow and so forth), despite the design of both the window-destruction and the Handle Manager of avoiding just that.

xxxDestroyWindow has two operation modes, a) actively by user-mode API invocation, b) from object unlocking. All over the code the developers mind this distinction, which is oversimplified here.

When object unlocking invokes the relevant destruction function of an object, it specifies it's running now in an atomic mode – implying to the rest of the win32 kernel code to not call back to user-mode or not even release the user "critical section", because this code block is critical for win32k "interrupts"...

The indication that the operation mode is now 'atomic' is done by incrementing a global named 'gdwInAtomicOperation' once the critical section begins; and decrementing the global once it ends (just like a sort of a scope-guard). Every time the win32k is about to callback to user-mode, it checks whether it is okay to do so by verifying this global value is clear. However, if it's set (atomic mode is on), it will proceed to check another global value 'gdwExtraInstrumentations', which is initialized to 0 upon boot and never changes. If the latter is on, there will be a bugcheck immediately, but fortunately we're good. These atomic checks happen before all types of callbacks to user-mode.

See markers in the picture below.

```
xxxClientFreeWindowClassExtraBytes proc near

var_38= qword ptr -38h
var_28= qword ptr -28h
var_20= qword ptr -20h
arg_0= byte ptr  8
arg_8= byte ptr  10h
arg_10= byte ptr  18h
arg_18= byte ptr  20h

; FUNCTION CHUNK AT 00000001C0177404 SIZE 00000030 BYTES

push    rbx
push    rdi
sub     rsp, 48h
mov     rbx, rcx
mov     rdi, rdx
mov     rcx, gs:188h
call    W32GetThreadWin32Thread
mov     r8, [rax+1D0h]
mov     rax, [rbx+28h]
sub     rax, r8
mov     [rsp+58h+var_20], rdi
mov     [rsp+58h+var_28], rax
mov     rax, cs:__imp_gdwInAtomicOperation
mov     edx, [rax]        ; BugCheckParameter1
test    edx, edx
jnz     loc_1C0177404
```

```
; START OF FUNCTION CHUNK FOR xxxClientFreeWindowClassExtraBytes

loc_1C0177404:
mov     rax, cs:__imp_gdwExtraInstrumentations
mov     ecx, [rax]
test    cl, 1
jz      loc_1C008CA90
```

```
loc_1C008CA90:          ; this
lea     rcx, [rsp+58h+arg_8]
call    ReleaseAndReacquirePerObjectLocks::ReleaseAndReacquirePerObjectLocks(void)
lea     rcx, [rsp+58h+arg_0] ; this
call    LeaveEnterCritProperDisposition::LeaveEnterCritProperDisposition(void)
mov     ebx, 7Ch
mov     ecx, ebx
call    cs:__imp_EtwTraceBeginCallback
nop     dword ptr [rax+rax+00h]
lea     rax, [rsp+58h+arg_10]
mov     ecx, ebx
lea     r9, [rsp+58h+arg_18]
mov     [rsp+58h+var_38], rax
lea     r8d, [rbx-6Ch]
lea     rdx, [rsp+58h+var_28]
call    cs:__imp_KeUserModeCallback
nop     dword ptr [rax+rax+00h]
mov     ecx, ebx
call    cs:__imp_EtwTraceEndCallback
nop     dword ptr [rax+rax+00h]
lea     rcx, [rsp+58h+arg_0] ; this
call    LeaveEnterCritProperDisposition::~LeaveEnterCritProperDisposition(void)
lea     rcx, [rsp+58h+arg_8] ; this
call    ReleaseAndReacquirePerObjectLocks::~ReleaseAndReacquirePerObjectLocks(void)
add     rsp, 48h
pop     rdi
pop     rbx
retn
xxxClientFreeWindowClassExtraBytes endp
```

```
and     [rsp+58h+var_38], 0
xor     r9d, r9d         ; BugCheckParameter3
xor     r8d, r8d         ; BugCheckParameter2
mov     ecx, 160h        ; BugCheckCode
call    cs:__imp_KeBugCheckEx
```

4.b Calling back to user-mode from smashable sites

The goal is to find another feature/mechanism that can be reloaded on a zombie object without increasing its refcount and that eventually calls back to user-mode even though it's prohibited by design.

In a recent version of Windows, the implementation of the **extra window bytes** (see WNDCLASS [6] structure in MSDN for more information) section was changed to be implemented through user-mode. Probably due to previous known attacks in related code.

The new implementation required two new callback functions in user-mode: xxxClientAllocWindowClassExtraBytes and xxxClientFreeWindowClassExtraBytes and they are called correspondingly from xxxCreateWindow and xxxFreeWindow (the workhorse of xxxDestroyWindow).

Once a window is getting created inside the kernel, xxxCreateWindow calls back to user-mode by invoking xxxClientAllocWindowClassExtraBytes. The corresponding user-mode function in user32 DLL just allocates memory with the given size from kernel (actually that's the WNDCLASS.cbWndExtra we set) and returns that pointer to kernel.

```
PWND xxxCreateWindowEx(…)
{
  …
  PVOID ptr = xxxClientAllocWindowClassExtraBytes(pcls->cbWndExtra);
  if (NULL == ptr)
  {
    status = insufficient-resources;
    goto error_handling;
  }
  pwnd->extraWindowUMPtr = ptr;
  …
}
```

All we have to do is to hook the client-alloc-window-class-extra-bytes callback function in user-mode, and once the kernel calls us back, we DestroyWindow(), making this window becoming a zombie, and return to kernel. The kernel will continue its execution and reload this window with the user-mode pointer. Just like the zombie reloading pattern we adequately followed so far.

The problem here is that, this time, we're trying to reload a zombie from inside its own window creation functionality. And xxxCreateWindowEx is sensitive to the state of the window it creates, and if it sees at certain points, as it advances, that the window was destroyed, it stops the creation process, frees the window (which cleans up our user-mode allocations too) and returns NULL. Bleh.

Once we call DestroyWindow in the middle of CreateWindow, it will still return NULL, as it will think the window creation failed. However, since we managed to add more references to it just before we destroyed it, we know it exists as a zombie, but that's just a technicality to show the complexity of this new attack. Also finding the HWND from the client-alloc-window-class-extra-bytes hook stub was tricky as it's not passed to user-mode as an argument, but it's out of scope, see relevant ultimate-reloading POC for the technique implementation.

We got half a 'reload' working by now, and we merely need to find a way to make the window remain loaded with the user-mode pointer, or to find a way to stop xxxCreateWindow in the middle, until the moment it's smashed, which will then use the user-mode pointer, as required.

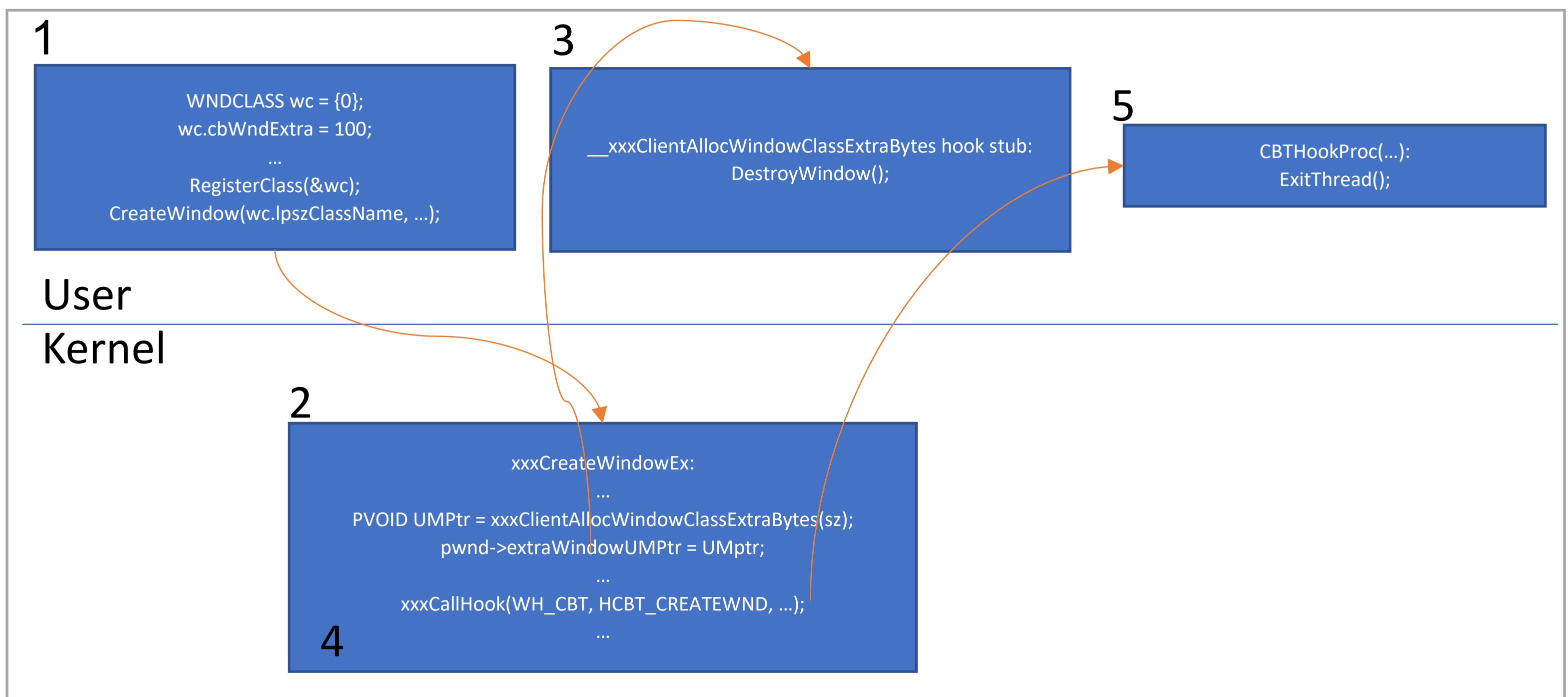
And this magic is possible by invoking ExitThread()!

To keep things simple, we won't dive into how thread termination works in NTOS and win32, but they obviously work together, and there are many quirks there too.

What we need to know is that the win32k counterpart (xxxDestroyThreadInfo) of the thread termination iterates over all the UI objects it owns, and destroys them, *but only if they're not destroyed already* (done by DestroyThreadsObjects), and then execution ceases on this thread, thus xxxCreateWindowEx will stop in the middle (not calling xxxFreeWindow to reverse our effect) and now we really got a half alive or dead zombie window!

Meaning that, after we do the extra-bytes pointer reloading technique, we have to return to kernel so the effect will be recorded on the window object, storing the user-mode pointer, and then precisely in the next callback to user-mode from the window creation, that's the CBT hook for HCBT_CREATEWND message, we call ExitThread, which leaves the zombie window, intact, as is.

This is how the flow happens:



At last, we managed to create a zombie window that once it's smashed, it **returns to user-mode right in the smashable site**, because the xxxFreeWindow (see stack trace below) function sees that the UMPtr isn't NULL, so it needs to callback to user-mode (to __xxxClientFreeWindowClassExtraBytes, which we can hook) to free that memory block. And that happens despite the fact it's an already destroyed zombie window, and this is how we get a *synchronization primitive* for our attacks, because from user-

mode, we can now easily manipulate and destroy any victim objects as we wish, and allocate new ones, etc.

This is how the beautiful stack trace looks like when exploiting the user-mode callback technique:

```
00 ffffb408`a65af700 fffffb8a`13676bfe win32kfull!xxxClientFreeWindowClassExtraBytes+0x8c
01 ffffb408`a65af760 fffffb8a`1368a5b7 win32kfull!xxxFreeWindow+0x22a
02 ffffb408`a65af880 fffffb8a`13a41745 win32kfull!xxxDestroyWindow+0x377
03 ffffb408`a65af980 fffffb8a`139cd6a9 win32kbase!xxxDestroyWindowIfSupported+0x25
04 ffffb408`a65af9b0 fffffb8a`139d69a5 win32kbase!HMDestroyUnlockedObject+0x69
05 ffffb408`a65af9e0 fffffb8a`136d7a51 win32kbase!ThreadUnlock1+0x95
```

By the way, the original reloading technique is still needed in conjunction with this new technique, because in order to smash an object, it's still required to reload it only once with a specific feature, stacked with this technique. Thus, forcing the controlled smashing point to get back to user-mode.

4.c Mixing zombie reloading with return-to-user-mode primitive

The advantage of this new technique is that now it lets us separate the victim object of the UAF attack from the target zombie object, as it's not necessary to link the two anymore. From now on it's possible to destroy any unlocked objects we wish, just directly from user-mode once we know we're called from a smashable site from the kernel.

Since the ultimate reloading technique requires to call ExitThread, it makes mixing the zombie reloading techniques a lot harder in reality. As we will need another thread to launch the rest of the attack - the actual zombie reloading inside the kernel and smashing.

If we recall how the previous timers attack works, we will know that eventually there are two timers involved, but basically now with the possibility of going back to user-mode in the middle, we can create a whole new different attack scenario on the target DestroyThreadsTimers function, we still need two timers, but not to chain them.

The attack's end goal is the same – to create a timer object UAF. For the new attack implementation to succeed it's enough to reload the zombie window with a single timer. Rather now, instead of chaining a second timer object, we simply chain the return-to-user-mode primitive. Once we're called in user-mode from the smashing site, we can free another arbitrary timer that was created in advance, and that will cause the UAF.

The following diagram flow shows the new implementation using two threads in parallel:

Main thread #1

User

WNDCLASS wc = {0};
wc.cbWndExtra = 100;
…
RegisterClass(&wc);
CreateWindow(wc.lpszClassName, …);

Kernel

xxxCreateWindowEx:
…
PVOID UMPtr = xxxClientAllocWindowClassExtraBytes(sz);
pwnd->extraWindowUMPtr = UMptr;
…
xxxCallHook(WH_CBT, HCBT_CREATEWND, …);
…

User

__xxxClientAllocWindowClassExtraBytes hook stub:
g_hwnd = …;
CreateThread();
Wait-for-phase-2;
DestroyWindow(); // Owner must call

User

CBTHookProc(…):
ExitThread();

#5 – That's the last user-mode code that runs after everything else is over, and calls ExitThread to trigger the exploitation!

1

Thread #2

User

Thread 2 proc:
SetWindowsHookEx(WH_SHELL, …);
FlashWindow(g_hwnd, …);
// --------------------
ExitThread();

5

Kernel

xxxFlashWindow:
…
xxxCallHook(WH_SHELL);
…
InternalSetTimer();

4

2

3

User

ShellHookProc(…):
Signal-phase-2-begins;
Wait-for-Thread#1 and window-destruction

Edge #1 - The main thread does the same thing as it did before in order to reload the return-to-user-mode feature, but with one modification - in the client-alloc-window-class-extra-bytes stub in user-mode it will create the second thread and wait for it to be inside the kernel too (for more references on the zombie).

Edge #2 – The main thread is busy looping until the second thread is ready for zombifying the window (i.e. calling DestroyWindow on the object). This is critical as if we destroy it too soon, other APIs on the second thread won't be able to access the object anymore.
Once the second thread started launching the attack using FlashWindow, we're called back to user-mode via a shell hook callback, and signaling the first thread we're ready.

Edge #3 – The second thread is now waiting for the main thread to kill itself after zombifying the window with the return-to-user-mode feature reloaded.

#4 – The shell hook callback is over because the window and main thread are destroyed, and now it resumes inside the kernel back in xxxFlashWindow to set a timer on the zombie window. Once xxxFlashWindow returns to user-mode, there will be a single reference from a timer object on this window.

#5 – The second thread procedure can then resume to call ExitThread which will call the DestroyThreadsTimers to trigger the exploit, this time with return-to-user-mode!

Once the zombie window will be freed from FreeTimer, it will call back to the client-free-window-class-extra-bytes stub, where we can kill as many other timers as we wish, that we created in advance, and will now make the 'nextPointer' point to a freed timer object…

Note that this section showed the concept of how to mix primitives, however in reality, this specific mixing won't work as win32k thread destruction (where the UAF takes place) will block attempts to callback to user-mode. And yet we decided to stick with this example for brevity, instead of introducing a whole new bug. (POC #5 - FreeSPB shows this concept).
Having said that, there are more caveats ahead:

1. Converting a single thread zombie reload to a multiple-threads reload is not as simple as it seems, because it depends on the functionality of how kernel calls back to user-mode.
   For example - in the previous attack, we used a window-message callback. That wouldn't work in our new multi-threaded scenario, as a window-messaging will call back to user-mode to the thread owning the window, which is the first thread, which isn't good for this new attack according to the described flow, so a shell-hook was used instead!
   Sometimes if you don't have possible alternatives, mixing can't happen, meaning that not all attacks are convertible to this new method.
2. An object normally must be destroyed from its owner/creator thread.
   E.g. calling DestroyWindow is thread sensitive. However, we don't violate this constraint.
3. In practice, once a thread is destroyed and it has an object that has a positive number of references (say, by objects of other threads), the object ownership is changed to the system (that's CSRSS in win32k's case). In our current example we ignore this fact for brevity but sometimes, there's a very complicated way to deal with, if required (shown in POC #12).
   (Note that this CSRSS ownership behavior was exploited in POC #11 to attack CSRSS itself.)
4. Once win32k thread destruction (xxxDestroyThreadInfo) starts, it sets a flag indicating to block all attempts to callback to user-mode.

4.d Unlocking new opportunities

With the ultimate reloading feature, we can now attack at least a dozen new vulnerable locations that previously couldn't be exploited by the chain-effect, because sometimes we couldn't link or get a last reference situation on a sub-resource in a smashable site.

The following is a pseudo code of a new vulnerable function as an example:

```
void UnlockNotifyWindow(pmenu)
{
  for (pitem = pmenu->items; pitem != &pmenu->items[pmenu->cItems]; pitem++)
  {
    if (NULL != pitem->psubmenu)
    {
      UnlockNotifyWindow(pitem->psubmenu);
    }
   }
  }
  HMAssignmentUnlock(&pmenu->notificationWnd);
}
```

Once a 'notificationWnd' is smashed inside the **recursion**, where some menu in the menus-tree isn't temporarily-locked at all, the kernel will call back to user-mode where we destroy all relevant menus, and back in the kernel it will continue iterating over freed menu objects…

This attack concept is generic in case there's a smashable site and a victim object that can be fully released independently from user-mode. Eventually, if the victim object to be UAF'ed is still referenced by other objects and unlinking isn't possible in some scenarios, then this attack isn't viable.

5 Exploitability

This bug class affects all versions of Windows, both 32 and 64 bits. Even though the research was done on Windows 10 64 bits. All the POCs are reproduced on WIP as the time of writing and work from a guest account.

The ultimate reloading technique probably doesn't work on older versions of Windows, but the same technique with the ExitThread can possibly be exploited to achieve the user-mode callback eventually by other means (xxxSendMessage , etc).

Windows 10, thanks to a recent mitigation (TypeIsolation implemented by NSInstrumentation class) that memsets objects and isolates them to their own pool, successfully blocks exploitation for some object types like a window.

However, unfortunately that mitigation doesn't cover other data structures in the kernel that are still susceptible to this attack, such as timers, menus, or thread-queues, etc.

On latest Windows 8.1, a released window object is still fully exploitable.

6 Possible Mitigations

Microsoft works hard to keep patches and code backward compatible, making it very hard to suggest a *good patch* without the *full* understanding of such a complex system.

However, it's still worth documenting potential mitigations as the fixes required to overcome this bug class are divided to a few different parts, because few variants exist, and as each requires different treatment.

Also, as some might believe it's better to be proactive and harden the code base in advance.

6.1 Options to mitigate the chain-effect of unlocking a zombie

1) First and obvious fix is to go all over the code base and add extra ThreadLock and ThreadUnlock pairs in suspected smashable sites. This is option is hard to pursue, because there's a high risk of missing such sites, but each patch is laser focused. However, given the pattern of how a smashable site is defined, it's possible to write an automation script (e.g. with Semmle) to find such locations, assuming all types-information exist in the code base, this can be a valid option. Yet it's not future-compatible with new code.
2) A kernel function that can potentially manipulate a zombie object should check first if the window is destroyed and fail or ignore the operation. As was shown earlier in xxxFlashWindow or xxxSetWindowStyle, after every callback to user-mode, the UI object will have to be checked for destruction-status.

   Today, it's already done inside xxxSetParentWorker function, for instance, and it really blocked a few potential smashable sites. This option is possible in conjunction with the other options, only for selected key functions.
3) Create a new mechanism that *defers* the destruction of all sub-resource objects that should get destroyed in an unlock context by the chain-effect (can determine when it's the case by the atomic operation global, discussed earlier) to the time the user "critical section" is released.

   This option is probably the safest and most generic solution. And reminds of similar new mitigation done for objects such as menustate, popup, and menu in recent versions of Windows.

   However, there's a tiny very technical caveat: assuming an object should get destroyed in a legitimate chain-effect, then with this fix, its destruction will be deferred – the only difference now (except memory usage) is that its handle-index isn't back to the freed-pool of handles in the Handle Manager.

   The advantage is clear as a generic solution will block this bug class for good and either old/hidden or futuristic smashable sites are not a threat anymore. Performance is a non-issue as the list of deferred objects will be most of the time empty anyway. And similar mechanisms are already deployed for other objects in different variations…
4) The solution suggested in option #3 is good, but unfortunately the win32k code is a bit buggier than that in rare cases. For instance, inside xxxSetCapture, an object is being unlocked and used afterwards anyway. If it were the last reference, then the code is now using a stale pointer (attached in a xxxSetCapture POC).

Perhaps such a case can be fixed manually once, but it can still happen in new code, or being hidden somewhere in the existing huge code base. Therefore, it might be necessary to enhance option #3 to defer only the destruction of the zombie object itself (and the rest [sub-resources] will follow automatically).

6.2 Window creation and destruction suggested fixes

1) xxxDestroyWindow should cleanup itself even in logically state-locked situations like in xxxReleaseCapture or FreeSPB/FindSPB. Technically, both functions are called, but don't do any work as their own state machine decided to block any changes. This is bad in case the active window is really being destroyed.
2) xxxDestroyWindow should fully clean up after itself all sub-resources like forgotten ones such as carets, self DDE-server windows, etc.

   Technically, the caret is cleaned very early in the flow of xxxDestroyWindow, but there are many callbacks to user-mode that give the opportunity to re-create a new caret for the window (as was shown earlier in attacking xxxCreateCaret thread-queue).

   The guideline should be to clean all sub-resources after all callbacks are done.
3) xxxFreeWindow should guard the call to run once xxxClientFreeWindowClassExtraBytes like WM_FINALDESTROY.
4) xxxCreateWindow should call xxxFreeWindow on zombie if it's about to return NULL.

6.3 Thread-queue suggested fixes

Thread-queues are problematic, because a function like AttachThreadInput, which is documented, can be called from user-mode in an unsynchronized fashion with DestroyWindow, and doesn't need the zombie reloading condition (as was explained earlier). Making it an easy target for attack.

Given how AttachThreadInput API works, thread-queues get reallocated and are not directly a sub-resource of a window but get manipulated during xxxDestroyWindow which can cause a UAF too. Even if a destruction of sub-resources is deferred, the thread-queue is still susceptible to our attack, because it's not a sub-resource per se (the Handle Manager doesn't handle it), and therefore still requires a fix.

In addition, going all over the code base and either reload the thread-queue pointer after a smashable site, or just change it to a full pointer chain referencing (pti->pq->field) every time, is a simple task and highly efficient against this attack.

For example, in xxxCapture there's a thread-queue pointer referenced immediately after a ThreadUnlock that is exploitable as demonstrated in one of the POCs, as well as inside xxxCreateCaret...

6.4 Atomic operations bugcheck enforcing

Seeing the code differences between Windows 7, 8 and 10, it's possible to judge that Microsoft is going forward with enabling the code that will bugcheck once kernel returns to user-mode in inappropriate critical times (such as the context of unlocking an object and many other situations).

If this mechanism was enforced by now then the ultimate zombie reloading technique (returning to user-mode from a smashable site) could have been mitigated, or in other words, it's a very effective mitigation.

Also, another suggested hardening is that upon any NtUser syscall entrance, or inside the function that acquires the "critical section" (EnterCrit), it will check for the atomic operation status, and once it's on, it will either bugcheck too or just fail.

7 Vulnerabilities coverage

There are many instances of this bug class in the win32 kernel code base, and over 11 instances were exploited with a triggering POCs. There is way more bugs than the amount that was exploited, but the main point was already proved. The POCs were run and tested on WIP (17763.1.amd64fre.rs5_release.180914-1434).

Each exploitation really requires so much research effort just in order to be able to juggle around the permanent locks in order to release the last reference in a target smashable site where the attack is viable. It was mostly omitted in this paper to reduce complexity, and it's of less importance for describing the techniques.

Remember that 3 functions can cause smashing triggerable: ThreadUnlock and HMAssignmentUnlock, obviously, but also, HMAssignmentLock, which decrements the refcount of a *previous* object, if exists. According to sum of all x-refs there are over 1000 such calls, although most are invulnerable. Sometimes the vulnerability lies in the caller of one of these functions (like in the case of xxxCreateCaret).

A window object can destroy with it many things, like other windows, timers, carets, icons or cause reallocation of thread-queues, etc. Normally functions that do cleanups will unlock window objects and have the potential to be exploitable. Such as: FreeTimer, FreeSPB, zzzDestroyCaret, etc. which were covered in the POCs too.

The POCs are designed to only trigger the UAF without any next level exploitation. Sometimes a POC won't bugcheck, but the bug is triggered because it touches a freed memory, and it also depends whether Windows has the mentioned mitigations. Given the pool is changed to reset an object's space, with 0xdeadbeef or likewise, after it was freed, that would really assist in spotting such cases.

Bugs covered in POCs:

1. xxxMnOpenHierarchy window UAF – Covered in section 3.b
2. FreeTimer timer UAF – Covered in section 3.c
3. xxxCreateCaret PQ UAF – Covered in section 3.d
4. Ultimate reloading - return to user-mode from xxxFreeWindow of a zombie window – Covered in section 4.b
5. FreeSPB window UAF
6. xxxCapture window UAF
7. xxxCapture PQ UAF
8. zzzAttachThreadInput PQ UAF
9. xxxSendMinRectMessages DesktopInfo UAF (POC catches free block before kLFH existed)
10. UnlockNotifyWindow menu UAF via user-mode callback
11. CSRSS Arbitrary user-mode heap-free (side effect of #4)
12. Advanced FlashWindow - Covered in section 4.c

13. UnlockDesktopMenu NULL dereference (another logical bug, was used in POC #10 - exploitable on Win 7 and below)

Potential vulnerable smashable sites:

The next list contains highly suspicious smashable sites (some are *very hard* to smash) to show that the problem is across the board. This list is probably incomplete and might contain false positives too, and if the proposed generic mitigation is applied, it won't matter anyway. Some bugs might lead to UAF, others to info-leaks, or other low-level exploitation primitives, depending on the target vulnerable function's implementation. Our goal with all POCs was to show that such complicated scenarios are possible to exploit and therefore we highly believe that the following list is also susceptible to this bug class.

Items 1-9 name the functions that contain the vulnerability and the smashable object, so looking at the function's code should be relatively easy to understand how to craft the attack.
Items 10-14 use primitives (or very similar) of attacks that are proved in the POCs above, and now turning them to attack other locations in the code. For example, in POC #10 we craft a special menu that once it's destroyed, it returns-to-user-mode, and therefore one can apply that together with targeting victim objects that are used after any function that calls DestroyMenu in the kernel. Note that the victim objects must have zero-references or the last reference decref'ed by the parent object destruction, otherwise the attack won't work.

1. xxxMnKeyDown - smash sys-menu inside next-menu message handling
2. xxxQueryDropObject - smash cursor
3. xxxSetFocus - smash returned window before it's temporarily locked
4. xxxMouseActivate - smash window inside loop of WM_PARENTNOTIFY
5. xxxDragObject – smash window after WM_DRAGSELECT
6. xxxCapture - Once the PWND UAF is fixed (POC #2), there's still another PQ UAF bug hiding…
7. SetMenuItemInfo - smash menu-item with a submenu after previous submenu is unlocked
8. xxxQueryDropObject - smash cursor inside recursion
9. SendMsgCleanup - smash messages after calls to xxxReceiverDied
10. Smash potentially zero-ref'ed objects after calls to MnFreePopup
11. Smash potentially zero-ref'ed objects after calls to both Un/LockPopupMenu
12. Smash potentially zero-ref'ed objects after calls to zzzDestroyQueue (use POC #8 as base to craft zombie)
13. Smash new window inside LockMFMWFPWindow
14. Smash menu-item after calls to MNFreeItem (user POC #10 to craft zombie)
15. Advanced vulnerability - xxxFreeWindow - HDC re-allocation post HDC cleanup, via user-mode callback of owner window smashing, and before window is marked for destruction…

8 Conclusion

In this paper we showed a new bug class that affects the win32 kernel code base of latest Windows 10 and probably has over a few dozen vulnerabilities that eventually lead to a memory use-after-free.

New exploitation techniques were developed to be able to attack the vulnerable locations and with high-probability succeed in the use-after-free exploitation.

An existing windows mitigation technique, called TypeIsolation, makes exploitation of some of these vulnerabilities harder and it depends on the object type. However, attacking older Windows versions with the same object type is still fully viable.

Other non-isolated objects are still vulnerable to the attack and are fully exploitable, even from a guest account.

The bug class proved to be hard to fix, and requires a few different solutions combined for full mitigation.

9 Greetings

Big thanks to the following reviewers to make this paper readable:
Ariel Shiftan, Assaf Segal, Liran Alon, Udi Yavo, Daniel Goldberg, and to a friend who preferred to stay anonymous.

To CC as I promised I'd do better.

References:

[1] Thomas Garnier, http://uninformed.org/index.cgi?v=10&a=2
[2] Tarjei Mandt, https://media.blackhat.com/bh-us-11/Mandt/BH_US_11_Mandt_win32k_WP.pdf
[3] MSDN, https://docs.microsoft.com/en-us/windows/win32/sysinfo/user-objects
[4] Raymond Chen, https://devblogs.microsoft.com/oldnewthing/20150918-00/?p=91561
[5] MSDN, https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-attachthreadinput
[6] MSDN, https://docs.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-wndclassa