Exploiting an Unbounded memcpy in Parallels Desktop

A Pwn2Own 2021 Guest-to-Host Virtualization Escape

Jack Dates May 19, 2022

This post details the development of a guest-to-host virtualization escape for Parallels Desktop on macOS, as used in our successful Pwn2Own 2021 entry. Given privileged code execution in the guest (i.e. via kernel module), the exploit obtains code execution in the Parallels application process running on the host system.

After providing a brief look at the approach I took towards evaluating Parallels and exploring some of its relevant attack surface, the remainder of the post will demonstrate how we were able to reliably exploit an unbounded `memcpy(...)` corruption style vulnerability to achieve arbitrary code execution.

A demo of the exploit running inside the guest popping calc on the host

Recon

Parallels Desktop is a virtualization product for macOS, comparable to other virtualization solutions such as VirtualBox or VMWare. It acts as a hypervisor to allow running guest operating systems (e.g. Windows, Linux) on a macOS host *'all without rebooting!'* (part of their marketing byline).

Parallels Desktop is a consumer-oriented virtualization solution on macOS systems

My primary motivation for taking a look at Parallels was due to its sudden inclusion in Pwn2Own 2021. Prior to this, nobody at RET2 had actually used Parallels before, let alone tried poking at it for vulnerabilities.

When faced with a new target, it can be a helpful first step to see what previous work already exists, be it writeups, bugs, exploits, or other relevant documentation/materials. At the time (early 2021), public research on Parallels was pretty scarce, save for one ZDI blog post and a handful of ZDI advisories. Since Pwn2Own, a few more resources [1, 2, 3] have trickled out, but the number of public works is still limited.

Identifying Guest-to-Host Attack Surface

After creating a new VM and spending some time getting acquainted with Parallels' features, some of the more interesting functionality that immediately stood out was the numerous "tools" made available to the guest VM.

Guest tools (or Parallels Tools) implicitly provide most of the sharing features offered by Parallels: things like drag-and-drop, copy-pasting between host and guest, shared folders, and dynamic screen resolution.

- *At this point, it was quickly apparent that some of Parallels' by-design shared folder behavior could be used for a guest-to-host escape. As part of the Pwn2Own meta-game, I opted to ignore these issues fearing a possible collision (it seemed at least two other teams ended up finding these same issues!)*

My focus shifted to the host-side implementation of these tools, presuming it was likely that the guest directly controlled input to that part of the code. To pursue this route, a binary to reverse would be helpful. By spawning a few VMs and looking at the system process list, we can infer `prl_vm_app` is the binary that runs for each guest.

Investigating the Parallels Host Process

While evaluating the Parallels worker process running on the host, we are most interested in identifying the parts of the code to which we can directly control input. The `prl_vm_app` Mach-O was massive, and putting in some reversing effort starting at the entrypoint did not prove immediately helpful.

I chose to instead look at a legitimate use case of the guest tools, with the hope that this would help identify where to start looking in the host `prl_vm_app` binary. The guest side of the tools implementation takes the form of several kernel modules, which act as a client for the interface provided by the host.

Fortunately, for Linux guests, Parallels provides source code for the kernel modules, which must be built from source depending on the specific Linux distribution. By reading the source, we can infer:

- how the guest provides input to the host side of the tools implementation
- how this interface is intended to work

There are five kernel modules, each with separate source directories:

- `prl_tg`: interfaces with the “toolgate”, a generic way to pass data to operations identified by opcode
- `prl_eth`: virtual network
- `prl_fs`: shared folders
- `prl_fs_freeze`: a suspend/resume helper dealing with filesystem structures
- `prl_vid`: video, potentially with some sort of DRM

One of the header files for `prl_tg` included opcode definitions for a multitude of toolgate operations, and based on the large number of them, this interface appealed the most.

// mouse pointer requests (from PCI video)

#define TG_REQUEST_SET_MOUSE_POINTER 0x100

#define TG_REQUEST_HIDE_MOUSE_POINTER 0x101

#define TG_REQUEST_DISABLE_MOUSE_POINTER 0x102

// multi-head support (from PCI video)

#define TG_REQUEST_VID_QUERY_HEADS 0x110

#define TG_REQUEST_VID_ENABLE_HEAD 0x111

...

// shared folders requests

```
#define TG_REQUEST_FS_MIN 0x200

#define TG_REQUEST_FS_GETLIST 0x200

...

// CPTool requests

#define TG_REQUEST_CPTOOL_MIN                  0x8010

#define TG_REQUEST_CPTOOL_HOST_TO_GUEST_COMMAND 0x8010

#define TG_REQUEST_CPTOOL_GUEST_TO_HOST_COMMAND 0x8011

#define TG_REQUEST_CPTOOL_MAX                  0x8012

// Shell integration tool

#define TG_REQUEST_SHELLINT_MIN           0x8020

#define TG_REQUEST_SHELLINT_ENABLED       0x8020

...

// OpenGL

#define TG_REQUEST_GL_VERSION       0x8130

#define TG_REQUEST_GL_CREATE_PROCESS  0x8131

...
```

A sample of some of the opcodes defined in prl_tg/Toolgate/Interfaces/Tg.h

From the guest tools source code, we can see that the guest kernel invokes toolgate operations by writing the physical address of a request structure to an I/O port. The request structure contains an opcode, size, and optionally inline data and/or "out-of-line" buffers.

While certainly useful context, none of the information we've gathered from the guest tools source code can be taken as ground truth – It is merely a suggestion of intent. We'll need to find the host-side implementation of the toolgate to see how things really (don't) work.

Locating the Toolgate Entrypoint

Shared folders were something easily poked at (i.e. with simple command-line file operations in the guest), and internally appeared to be implemented with toolgate requests. Finding the host implementation of shared folders seemed a promising path to finding the overall toolgate implementation.

The following procedure can be used to locate the `prl_vm_app` function responsible for shared folders requests:

- create a guest with tools installed; default config shares the home directory at `/media/psf/Home`
- on the host, attach (e.g. with lldb) to the `prl_vm_app` process and set a breakpoint on `unlink`
- in the guest, create and delete a file in the shared folder, hitting the breakpoint
- look at the backtrace

Matching up the backtrace functions with those in your disassembler/decompiler of choice, you should see a function with a switch-case over the various toolgate opcodes for shared folders requests.

However, this function solely handles shared folders requests, and reversing deeper into the backtrace does not bring us to the initial toolgate entrypoint as we hoped... The shared folders requests are handled on a separate thread; the requests are taken from a queue and handled asynchronously. Our search will now turn to finding where the queue is populated.

This can be done with watchpoints:

- set a breakpoint where the requests are taken off the queue
- in the guest, trigger a shared folders toolgate request, hitting the breakpoint
- note the address of the array used for the queue slots, set a watchpoint on the next array slot
- in the guest, trigger another shared folders request, hitting the watchpoint
- look at the backtrace

The watchpoint should trigger on a thread named `VCPU0` (or potentially another "virtual CPU" thread if there are multiple cores), and reversing through the backtrace should eventually reveal the toolgate entrypoint, where the guest-supplied request structure is parsed and interpreted.

Toolgate Protocol and Handlers

Using the guest tools source as a guide, we can reverse engineer how toolgate requests are parsed and handled.

To interact with the toolgate, the guest writes a physical address to a `TG_PORT_SUBMIT` I/O port. Other ports are indicated in the guest tools source (these port numbers would be offset by a base port):

```
/* Symbolic offsets to registers. */

enum TgRegisters {

    TG_PORT_STATUS  = 0,
    TG_PORT_MASK    = 0,
    TG_PORT_CAPS    = 0x4,
    TG_PORT_SUBMIT  = 0x8,
    TG_PORT_CANCEL  = 0x10,
    TG_MAX_PORT     = 0x18
};
```

For example, the base port in the guest is usually `0x8000`, so to submit a toolgate request, the physical address of the request structure would be written to port `0x8008` with the `out` instruction.

The host then expects a `TG_PAGED_REQUEST` structure to be present at the guest physical address (again, structure definitions taken from guest tools source):

```
typedef struct _TG_PAGED_REQUEST {
    unsigned Request;
    unsigned Status;
    unsigned RequestSize;
    unsigned short InlineByteCount;
    unsigned short BufferCount;
    TG_UINT64 RequestPages[1/*RequestPageCount*/];
    // char InlineBytes[(InlineByteCount + 7) & ~7];
    // TG_PAGED_BUFFER Buffers[BufferCount];
} TG_PAGED_REQUEST;

typedef struct _TG_PAGED_BUFFER {
    TG_UINT64 Va;
    unsigned ByteCount;
    unsigned Writable:1;
    unsigned Reserved:31;
    // TG_UINT64 Pages[];
} TG_PAGED_BUFFER;
```

- `Request`: the “opcode” of the toolgate operation
- `Status`: a return value written by the host to indicate success, an error, or pending
    - Certain requests are asynchronous (and will be pending), as we saw with shared folders earlier. These can be canceled by writing the request’s physical address to the `TG_PORT_CANCEL` port.
- `RequestSize`: size of the variably sized `TG_PAGED_REQUEST`
- `RequestPages`: physical page frame numbers for each page in the `TG_PAGED_REQUEST` structure
    - Only really necessary if the request spans multiple pages. `RequestPages[0]` is somewhat redundant considering the physical address written to the I/O port
- `Buffers`: “out-of-line” data blobs, again with physical page frame numbers for each page of each blob
    - The name `Va` implying virtual address is a bit odd, but in reality only the page offset (`Va & 0xfff`) matters, the `Pages` array defines the actual location in guest memory by page frame number

The host is free to use the inline data and/or the buffers both as input and/or output. For example, a specific opcode may expect `Buffers[0]` to be an input buffer, and will use `Buffers[1]` as the output.

In `prl_vm_app`, the toolgate is represented by a `CPCIToolgate` class. One member is an array of registered handler entries containing a minimum/maximum opcode and the handler object itself. Upon receiving a guest request, this array is searched through to find the handler (if any) that handles the given opcode. Then a virtual call on the handler object does all the operation-specific work.

The members of this array will define our attack surface. Breaking in the debugger and dumping the handlers array, we can map opcode ranges to their handlers:

Opcode range (inclusive)	Handler C++ class name
0x200-0x23f	CSFoldersTool
0x8000-0x8001	CUTCommandsSender
0x8010-0x8012	CPTool
0x8020-0x8029	CSMTool
0x8030-0x8033	LayoutSyncHost
0x8040-0x8044	CSFoldersTool
0x8050-0x8050	Tg2OtgImpl
0x8200-0x8202	CGracefulShutdownHost
0x8210-0x8211	CToolsCenterHost
0x8220-0x8221	CSIAServer
0x8230-0x8231	CCoherenceToolServer
0x8301-0x8301	CWinMicroApp
0x8302-0x8302	CFavRunAppsHost
0x8304-0x8304	CDragAndDropBase
0x8320-0x8328	CSHAShellExt
0x8340-0x8347	CUIEmuHost
0x8410-0x8411	CSharedProfileTool
0x8420-0x8440	CInvSharing

```
| 0x8500-0x8503             | CDesktopUtilitiesHost   |
| 0x8700-0x8702             | CSFilterTgHandler       |
| 0x8800-0x8801             | CHostCEP                |
| 0x8900-0x8900             | CPrintingTool           |
| 0x9020-0x9021             | CLocationTool           |
| 0x9030-0x9030             | CEnergySavingTool       |
| 0x9040-0x9043             | CVSDebug                |
| 0x9050-0x9050             | AutoPauseHost           |
| 0x9060-0x9060             | VmHostname              |
| 0x9070-0x9070             | UniversalChannelHost    |
| 0x9080-0x9080             | CDynResHost             |
| 0x9100-0x9101             | CWinMicroApp            |
| 0x9110-0x9111             | VolumeControllerHost    |
| 0x9120-0x9124             | TimeSynchronizationHost |
| 0x9200-0x9200             | CToolbox                |
```

There seemed to also be a video toolgate completely separate from the standard toolgate (i.e. different virtual device, different I/O base port). This component wasn't investigated much, but contained the following opcode ranges dumped dynamically as before:

```
| Opcode range (inclusive) | Handler C++ class name |
| ------------------------ | ---------------------- |
| 0x100-0x11f              | VideoTgHandler         |
| 0x400-0x41f              | DirectXHandler         |
| 0x8100-0x811f            | VideoTgHandler         |
| 0x8130-0x813f            | OpenGLHandler          |
```

We can now focus our reversing efforts on the handlers' virtual methods for handling toolgate requests, code we know is reachable with arbitrary input. Based on the handler class names, these requests should do "interesting" things. The workflow for this is to pick a handler class, find its vtable (which will have a symbol name like `vtable `for'CDragAndDropBase`), then reverse the virtual handler function (for example, the function at offset 16 in the vtable).

The path forward from here to vulnerability discovery is mostly the standard reversing grind. Let's move on to the bugs found in some of these opcode handlers.

Bug 1: Megasmash (an unbounded `memcpy(...)`)

Opcode `0x8050` is handled by `Tg2OtgImpl`, seemingly a proxy between the toolgate and "OTG" (possibly Open Tools Gate). The proxy works something like this:

1. `Buffers[0]` contains a header of sorts. This includes the size of the OTG input request, and will contain the output return value and response size once handled
2. According to the input size from the header, the remaining buffers (starting at `Buffers[1]`) are "flattened" into a single byte array. This flattened array is treated as the OTG request to pass through.
3. An OTG opcode is fetched from offset 0 in the flattened blob, and the request is dispatched to the appropriate OTG handler for that opcode
4. The return value and response size are set in the header, and the response blob is "unflattened" into `Buffers[1]` onwards

The flattened request is represented by a `QByteArray`, a class supplied by the 3rd-party QtCore library, part of the Qt family. It is instantiated in the following way, using size from the input header:

QByteArray::QByteArray(&arr, hdr[2], 0LL);

And the respective QtCore implementation:

QByteArray::QByteArray(int size, Qt::Initialization)

```
{
    d = Data::allocate(uint(size) + 1u);
    Q_CHECK_PTR(d);
    d->size = size;
    d->data()[size] = '\0';
}
```

There is a pretty standard integer overflow here. If the provided `size` is -1 (`0xffffffff`), the size passed to `Data::allocate` will be 0.

Looking at the `allocate` implementation we can see that the following code will return a special empty singleton in QtCore’s data section, `qt_array_empty`:

```
QArrayData *QArrayData::allocate(size_t objectSize, size_t alignment,
        size_t capacity, AllocationOptions options) Q_DECL_NOTHROW
{
    ...
    // Don't allocate empty headers
    if (!(options & RawData) && !capacity) {
        ...
        return const_cast<QArrayData *>(&qt_array_empty);
    }
    ...
}
```

A `QByteArray` typically stores its data inline; or in other words, `d->data()` in the snippet above is effectively `(void*)d + sizeof(*d)` (a pointer to the address immediately following the declared structure fields). In the case of the singleton, the inline data would also be in the data section (although it has size zero).

Back in the OTG proxy, the provided buffers will be flattened by a loop, which copies one buffer at a time into increasing offsets of the newly constructed `QByteArray` (which is expected to have size `0xffffffff`, but is actually empty).

The overall effect of this loop can be summed up as:

memcpy(QtCore_data_section, controlled_data, 0xffffffff);

In theory, this lets us corrupt a significant portion of QtCore's data section. There is, however, the small problem of the excessively large copy being guaranteed to cause a fault (there will be a read-only mapping at some point after the writable data section).

These kinds of "wild copy" bugs are less than ideal but have been proven exploitable by researchers several times in the past [1, 2]. Sometimes it is possible to modify memory or program state during the copy loop to cut it short, which wasn't viable in this case. Another method is to race another thread using the corrupted data before the copying thread crashes, which I could not find a reliable method of doing.

We'll just keep this bug in our back pocket for now...

Bug 2: An Information Leak

This bug is less exciting, but provides a useful information leak to bypass ASLR in the host process.

The OTG handler for OTG opcode 10 appears to fetch various configuration settings for the guest. Different types of settings can be requested, the vulnerable type is 2. Here is a rough pseudocode interpretation for this type of request:

```
void otg_getdisablewin7logo(QByteArray *req, QByteArray *out) {
    QByteArray::resize(out, 0x90); // new bytes left uninitialized (docs)

    // write the first 0x20 bytes of out->data()
    memcpy(out->data(), req->data(), 0x10);
    if (CVmRunTimeOptions::isDisableWin7Logo(...))
        *(long*)(out->data() + 0x10) = 0x600000001;
    else
        *(long*)(out->data() + 0x10) = 0x1f00000000;
    *(int*)(out->data() + 0x18) = ...;
    *(int*)(out->data() + 0x1c) = 1;
}
```

This handler resizes the output `QByteArray` to size `0x90`, then proceeds to populate the first `0x20` bytes. The remaining `0x70` bytes are left uninitialized, and will be copied out to the guest by the OTG proxy.

This allows us to leak `0x70` bytes of uninitialized heap memory from the `prl_vm_app` host process, which can be repeated as necessary to obtain any leaks (heap or .text) we need.

Bug 3: Limited Stack Buffer Overflow

Opcode `0x8304` is handled by `CDragAndDropBase`, presumably to implement drag and drop behavior.

This handler uses “worker” objects, represented by `CDragDropWorkerBase`. An initial request must be made to create a new worker, and a handle to this worker is passed back to the guest for use in subsequent requests. As a side note, the handle was simply the heap address of the worker object, so was a limited information leak by itself.

Drag-and-drop requests have their own structure expected by the host, including a type field specifying what drag-and-drop operation to perform. A pseudocode snippet of the vulnerable code is reproduced below, preceded by some relevant structure definitions (struct interpretations are my own):

```
// input request from guest, completely controlled
struct dnd_blob {
    unsigned type;
    ...
    QChar qstr[1024];
    ... // 0x74 additional bytes
};
// internal message passed off to further layers
```

```
struct msg {
    ...
    QChar qstr[1024];
    ... // 0x68 byte gap
    CDragDropWorkerBase* worker;
};

void CDragAndDrop_process_blob(CDragDropWorkerBase* worker,
tg_req* req, dnd_blob blob) {
    switch (blob.type) {
        case 0x107:
            QString qstr = QString::fromUtf16(&blob.qstr, -1).normalized(); //
[1]
            ...
            struct msg msg;
            msg.worker = worker;
            memcpy(msg.qstr, qstr.utf16(), 2*qstr.size() + 2); // [2]
            ...
            msg.worker->queued_req = req; // [3]
            return;
    }
}
```

The guest-supplied string is supposed to contain a maximum of 1024 UTF-16 characters. At the first annotation `[1]` the length passed to `QString::fromUtf16(...)` is -1, indicating that the input should be treated as null-terminated to determine the actual length.

There is no check, however, that the input indeed contains a null terminator. The `0x74` additional bytes following the input (and potentially whatever is on the stack afterwards) may be interpreted as part of the string, exceeding the expected 1024 length limit.

At `[2]`, the string is copied into a local stack structure. This is where the overflow can occur. There is just enough data in the input `dnd_blob` structure to corrupt the `msg.worker` pointer with a controlled value, which is written to at `[3]`. This gives us a write of an uncontrolled value (the `req` pointer) to an arbitrary location.

In theory, if the contents of the stack after the `dnd_blob` structure were non-zero and somewhat controllable, this overflow could corrupt the return address or other stack data. Empirically, there seemed to always be zeroes following the `dnd_blob`, and this path was not pursued.

Exception Handling on macOS

It is likely possible the semi-controlled pointer write above would have been enough to finagle our way to code execution. Instead, our exploit will leverage this weaker bug to render the megasmash (bug #1) reliably exploitable.

The key realization to make is that when the megasmash inevitably causes a page fault, this doesn't necessarily result in the death of the process. As part of the Mach architecture of macOS, the page fault generates an exception (`EXC_BAD_ACCESS`), which is sent as a mach message to the registered exception port (at either the thread, task, or host level). The thread that generated the exception remains suspended until a response to the mach message is received (but other threads may continue to run).

Under normal circumstances, there is neither a thread-level or task-level exception port registered, and the host-level (i.e. system-wide) exception handling ends up killing the task. `prl_vm_app`, however, does register a task-level exception port using `task_set_exception_ports`, for 3 exception types: `EXC_BAD_ACCESS`, `EXC_BAD_INSTRUCTION`, and `EXC_ARITHMETIC`.

I Know the Pieces Fit

In terms of intended functionality, my guess is that Parallels intercepts these exceptions for crash reporting purposes. A separate thread is spawned with the sole purpose of, in a loop, reading messages from the exception port, handling them, and responding back.

What happens if this exception handling thread triggers an exception?

1. The exception handling thread enters the kernel exception delivery code, and is effectively suspended
2. A message is sent to the task-level exception port
3. The kernel waits for a response…

This response will never come, seeing as the thread intended by `prl_vm_app` to respond to the exception port message, is itself suspended due to an exception.

As a side note, a potentially more robust technique would have been to set the exception ports at the thread level (with `thread_set_exception_ports`), for each individual thread *except for* the exception handler. That way, exceptions generated by the handler will get forwarded to the default host-level machinery.

If we find a way to force the exception handler thread to fault, the megasmash suddenly seems a lot more exploitable:

1. Perform whatever setup so the exception handler will crash later
2. Megasmash the QtCore data section, which faults and generates an exception
3. Exception handler faults and generates an exception, leaving both threads suspended indefinitely
4. Wait for another thread to use the corrupted QtCore data section in an interesting way

Using our limited stack-based buffer overflow (bug #3), we can corrupt a pointer the exception handler will later attempt dereferencing, causing the thread to fault and suspend indefinitely.

This is simple to achieve in practice. When the exception handler is being initialized, prior to calling `task_set_exception_ports`, the pre-existing ports are retrieved (with `task_get_exception_ports`) and stored in a heap allocated structure. Later on, when the handler thread receives an exception message, one of its first actions is to reset the exception ports to the original ones using this structure. This heap pointer resides in the data section, and will serve as the target pointer we will corrupt.

Putting Everything Together

The final exploit then looks something like this:

1. Use the info leak (bug #2) to obtain the base addresses of `prl_vm_app` and QtCore
    - `prl_vm_app`: Experimenting by running the leak many times had a text pointer showing up relatively often. It turned out to be a function pointer passed to `CFRunLoopTimerCreate`
    - QtCore: The `CDragDropWorkerBase` objects contain `QList`

objects, which if empty, contain a pointer to `QListData::shared_null` (in the QtCore data section). By spraying many workers then deleting them, we can (eventually) get this leak

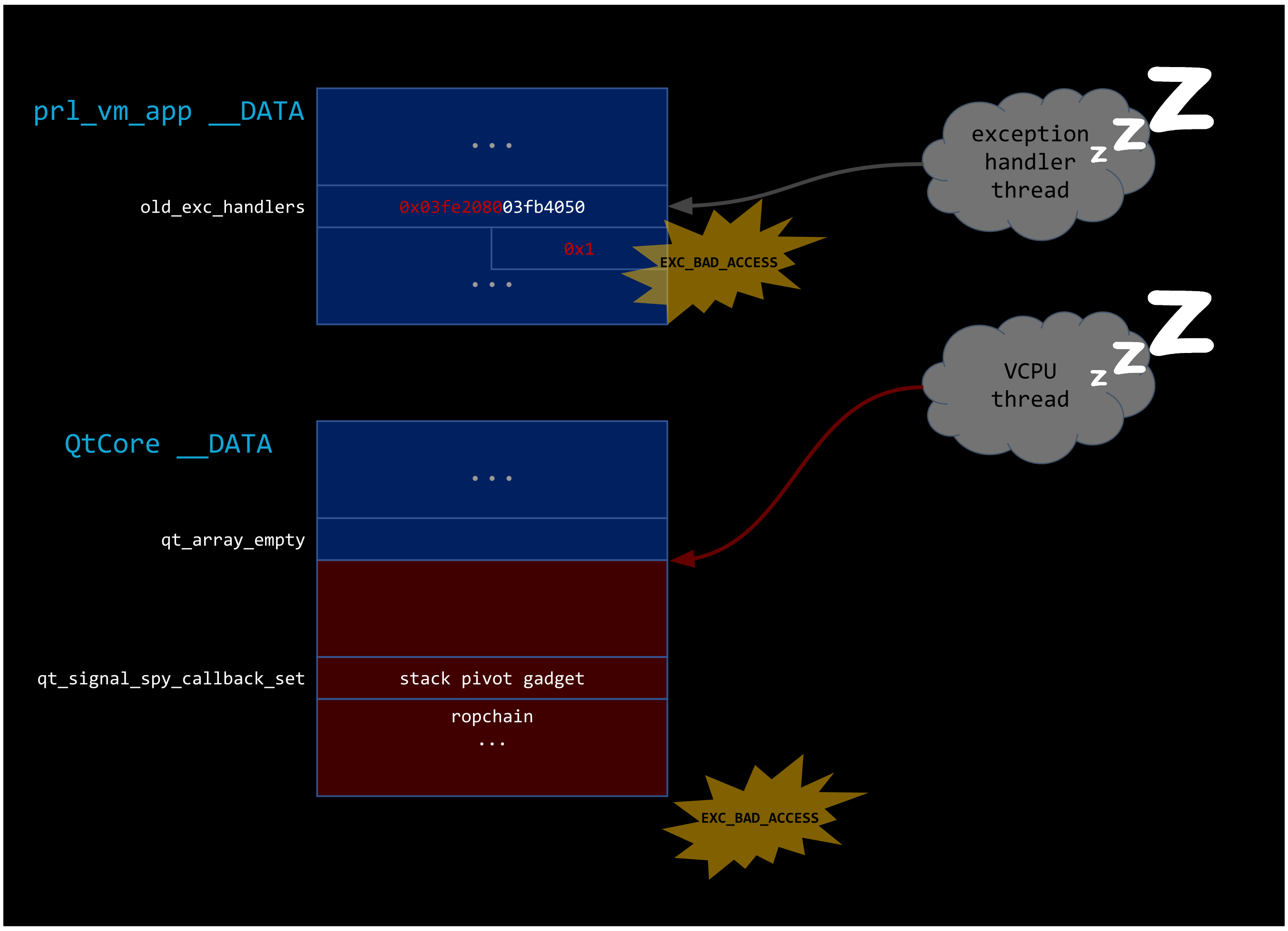
2. Use the uncontrolled pointer write (bug #3) to corrupt the pointer to the original exception ports
    - By misaligning the write 4 bytes into the 8-byte pointer, we can ensure the new value will be a non-canonical garbage address

3. Megasmash the QtCore data section (bug #1)
    - We target a function pointer `qt_signal_spy_callback_set` (a hook of sorts triggered often through normal operation of `prl_vm_app`) with fully controlled and carefully placed data, amidst our massive corruption of Qt's writable data segment
    - The megasmash will run off the end of the writable data segment triggering an `EXC_BAD_ACCESS`

4. Crash the exception handler thread
    - The `prl_vm_app` exception handler will attempt to restore the original exception ports from the heap-allocated structure. Since we corrupted this pointer with an invalid address in step 2, it will force the exception handler thread to fault.

5. Wait for another thread to use the corrupted function pointer
    - Stack pivot, short ropchain to call `system(...)`
        - `rcx` happens to be pointing into the corrupted QtCore data section, an `lea rsp, [rcx+0x30] ; ret` gadget kicks off the ropchain
    - Providing the string `open -a Calculator` to `system(...)` is sufficient to pop calc

Roughly, the steps described above which make up the final exploit are illustrated by the following animation:



A less dramatic video of the successful $40,000 exploit entry running live is available on ZDI's day two stream of Pwn2Own 2021. The full exploit code can be found here on our GitHub.

Bugfixes

The bugs used in this exploit were all patched by Parallels version 49187.

- Bug #3 (the stack buffer overflow) was patched with a check for a null word in the first 1024 UTF-16 characters
- Bug #2 (the information leak) was patched by calling `bzero(...)` after resizing the output array
- Bug #1 (megasmash) was patched by ensuring the size passed to

`QByteArray` is non-negative

  - While potentially a dangerous pattern, Qt stated that the QtCore integer overflow was not a bug in QtCore, instead that it is a misuse of the API to ask to create a structure with negative size
    - Interestingly, at the time, the master branch of QtCore did not exhibit the integer overflow due to a refactor in how the size parameter is passed around.

The confusion over whether Qt should actually fix the overflow further delayed the publication of this writeup.

For one reason or another, ZDI opted to drop Parallels from 2022's competition:

Bruno
@bkth_ · **Follow**

RIP Parallels 2021-2021

> **Zero Day Initiative** @thezdi
> Announcing #Pwn2Own Vancouver for 2022! We're celebrating the 15th anniversary of Pwn2Own with over $1 million in cash and prizes. Targets include Tesla, Zoom, Exchange, Teams, Ubuntu, and many more. Read the details at zerodayinitiative.com/blog/2022/1/12...

2:54 PM · Jan 12, 2022

42 Reply Share this Tweet

Read more on Twitter

Conclusion

In this post, we detailed a guest-to-host escape of the Parallels Desktop virtualization platform by chaining together three different vulnerabilities. More importantly, we were able to produce another meaningful (realworld) example of both surviving and then exploiting a ‘wild’ unbounded `memcpy(...)`

We believe that the techniques discussed here can be generalized and applied to successfully exploit similar issues that may otherwise appear impractical or impossible to exploit.