# How DopplePaymer Hunts & Kills Windows Processes

crowdstrike.com/blog/how-doppelpaymer-hunts-and-kills-windows-processes/

Shaun Hurley December 7, 2021


In a July 2019 blog post about DoppelPaymer, Crowdstrike Intelligence reported that ProcessHacker was being hijacked to kill a list of targeted processes and gain access, delivering a “critical hit.” Although the blog is now a couple of years old, the hijacking technique is interesting enough to dig into its implementation.

The hijack occurs when ProcessHacker loads a malicious stager DLL designed to exploit legitimate behavior. Once the process has been hijacked, the stager DLL is able to terminate processes, including those protected by Protected Process Light (PPL). To accomplish this task, it leverages ProcessHacker’s kernel driver, KProcessHacker, that has been registered under the service name KProcessHacker3. This blog delves into the details about how DoppelPaymer hijacks ProcessHacker and exploits KProcessHacker to kill a list of processes, including both antivirus (AV) and endpoint detection and response (EDR) applications.

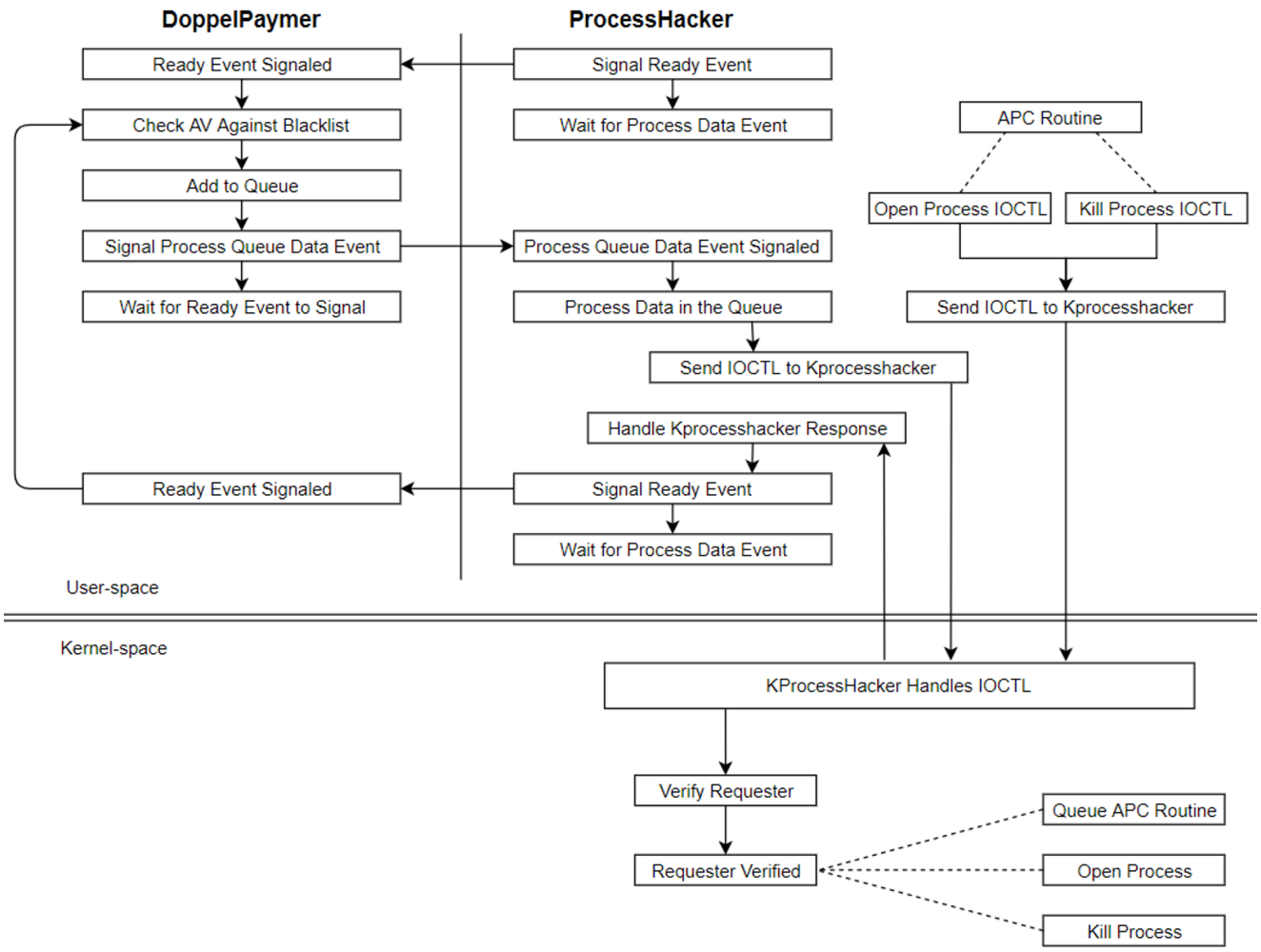


Figure 1. Architecture diagram to kill a target process

## Launching Process Hacker

To start ProcessHacker, DoppelPaymer writes the ProcessHacker executable, the KProcessHacker driver, and the malicious stager DLL into a subdirectory of `%APPDATA%`. Both the subdirectory name and the file names for the executable and driver are a unique string of alphanumeric characters. Once those two files have been written, one of the DLLs loaded by ProcessHacker has to be hijacked using a technique called “DLL search order hijacking.”

### DLL Search Order Hijacking

Similar to Dridex, DoppelPaymer uses DLL search order hijacking to exploit the DLL loading behavior for Windows processes. As the operating system PE loader loads a binary, it also needs to load the DLL files required for the PE to function. By default, MS Windows has a specific path it takes when looking for the DLL files to load. Windows checks for Windows

system DLLs in the same directory as the target binary before it checks the Windows system directories. A malicious process, in this case DoppelPaymer, can drop a malicious version of a DLL in that directory and it will be loaded by the target application.

To determine which DLL to hijack, DoppelPaymer walks the module name list in the Import Address Table (IAT) of the ProcessHacker binary. Each name is hashed with a CRC32 algorithm and compared against a hardcoded list of hashes (Table 1), and if a match occurs, the name is added to a list data structure. A random number generator is used to pick one of the three names out of the list.

| CRC32 | Filename |
|---|---|
| `0xd8946922` | `VERSION.dll` |
| `0x020da855` | `WINSTA.dll` |
| `0x3c55abe2` | `UxTheme.dll` |

Table 1. Mapping DLL name to CRC32 hash for search order hijacking

Once a DLL has been picked, the legitimate Windows version of the DLL is read into a memory buffer. This DLL is used as a template to build the malicious stager DLL. The file is written to the same subdirectory as the ProcessHacker executable with the same file name as the hijacked DLL.

## Creating the Process

DoppelPaymer passes two arguments to the ProcessHacker process: The first is the name of the KProcessHacker.sys driver, and the second is an integer that will be used for inter-process communication (IPC) between the DoppelPaymer and ProcessHacker processes.

```
C:\Users\ducksoup\AppData\Roaming\M28fPT\ibOLR 2LEQV0 161604546
```

Figure 2. ProcessHacker command line

### Setting Up the IPC Objects

Event handlers and section objects are used to communicate between the two processes. These objects allow DoppelPaymer to communicate directly with the stager DLL that is loaded inside the ProcessHacker process. The example handle values in Table 2 are used throughout the rest of this post when referencing these objects. These values vary with different executions of DoppelPaymer.

| Object Type | Handle Value | Purpose |
|---|---|---|
| Event Object | `0x120` | Notify data in queue |

| | | |
|---|---|---|
| Event Object | `0x11C` | Notify data processed |
| Section Object | `0x124` | Queue used to send process information to the stager DLL |
| Section Object | `0x128` | Contains the three events |

Table 2. IPC handles with concrete values from testing

For each section object, a view is mapped into process memory, so that DoppelPaymer is able to write data to the objects. The `0x124` object is the queue where the process information of the processes to terminate will be written. The other object, `0x128`, will contain the handle values of the other three objects: `0x120`, `0x11C` and `0x124`. For the stager DLL to access those three handles, DoppelPaymer needs to provide the `0x128` handle value to ProcessHacker.

Sticking with the example command line in Figure 1, the second argument to ProcessHacker is the section object handle `0x128` XORd against the same constant value (unique per binary) used throughout the lifetime of DoppelPaymer. For this binary, the constant is `0x9a1e2ea`. XORing `0x128` with `0x9a1e2ea` gives us the decimal value `161604546`.

After these IPC objects are created, and the second argument to ProcessHacker has been generated, `CreateProcessW` is called to launch ProcessHacker. Now DoppelPaymer has to wait for the stager DLL to initialize inside of the ProcessHacker process prior to establishing inter process communication. `NtWaitForSingleObject` is called for event handle `0x120`, and DoppelPaymer waits for that event to be signaled.

## Loading the Stager DLL

The stager DLL is loaded into ProcessHacker. Several initialization steps have to occur before the stager DLL can leverage KProcessHacker to kill processes:

- ProcessHacker's entry point needs to be modified to ensure that none of the startup routines for ProcessHacker execute
- The KProcessHacker service has to be initialized
- ProcessHacker and the stager DLL have to be verified as a valid client for the KProcessHacker service
- The IPC objects necessary for DoppelPaymer to communicate with the stager DLL need to be duplicated

After all four of these steps have been successfully completed, the stager DLL can start killing target processes provided by DoppelPaymer.

### Reaching ProcessHacker's Code Entrypoint Address

Once the process starts to load the stager DLL, the malicious code will start to execute, but if control isn't passed back to the OS to finish loading ProcessHacker, it will not be usable by DoppelPaymer. The loading process completes when the entry-point address of ProcessHacker is reached. To determine when the entry point is reached, the stager DLL will overwrite the entry point of ProcessHacker with the code in Figure 3.

```
.rdata:10006120             mov     eax, 94A351BBh
.rdata:10006125             push    0
.rdata:10006127             push    8FF4B5ACh ; Event handle
.rdata:1000612C             call    eax       ; NtSetEvent
.rdata:1000612E loc_1000612E:
.rdata:1000612E             push    0
.rdata:10006130             push    1
.rdata:10006132             push    0FFFFFFFEh
.rdata:10006134             mov     eax, 1DCB264Eh
.rdata:10006139             call    eax       ; NtWaitForSingleObject
.rdata:1000613B             jmp     short loc_1000612E
```

Figure 3. Entrypoint template code

This code is copied from the .rdata section of the stager DLL and is modified to represent the current process environment. Placeholders exist for the event handle and for the two Windows API functions used for the notification routines. The event used to signal that the entry point has been reached is created and copied to the `8FF4B5ACh` placeholder. The addresses for `NtSetHandle` and `NtWaitForSingleObject` are resolved and written to `94A351BBh` and `1DCB264Eh`, respectively.

Once the template is complete, `VirtualProtect` is called to set ProcessHacker's entry point to write-able, the entry point code is overwritten, and the original protection restored. The new entry-point code, in C, is shown in Figure 4.

```
//
// Signal entrypoint reached
//
NtSetEvent(entryPointReachedHdl, NULL);

while (1) {
    //
    //  Entrypoint thread will loop indefinitely
    //

    NtWaitForSingleObject(-2, 1, NULL);
}
```

Figure 4. ProcessHacker entry point infinite loop

The code in Figure 4 signals to the stager DLL thread that the entry point has been reached, and it continues in an infinite loop that calls `NtWaitForSingleObject`. Not only will this infinite loop let the stager DLL know when the entry point is reached, it also prevents

ProcessHacker from interfering with the stager DLL and prevents the ProcessHacker window from being displayed.

Now that the entry point is overwritten, the stager DLL spawns a new thread that initializes the KProcessHacker driver and sets the stage for killing AV processes. First, the thread calls `NtWaitForSingleObject` and waits for the entry point to be reached.

## Initializing the KProcessHackerDriver

The “entry point reached” event is signaled, and this thread can continue and initialize the KProcessHacker driver. The stager DLL has to create the KProcessHacker service and register the driver. The code to accomplish this task is essentially the same code used by the two ProcessHacker functions that can be found in the kph.c source code:

- `KphConnect2Ex`
- `KphConnect`

The code opens the service control manager in Windows and creates the KProcessHacker service under the name `KProcessHacker3`. The stager DLL passes the following arguments to the `CreateService` procedure:

```
CreateService(
     scmHandle,
     L”kprocesshacker3”,
     L”kprocesshacker3”,
     SERVICE_ALL_ACCESS,
     SERVICE_KERNEL_DRIVER,
     SERVICE_DEMAND_START,
     SERVICE_ERROR_IGNORE,
     //
     //   Path to kprocesshacker.sys driver file
     //
     L”C:\Users\ducksoup\AppData\Roaming\M28fPT\2LEQV0”,
     NULL,
     NULL,
     NULL,
     NULL,
     L""
);
```

The KProcessHacker service has been created and started and is ready to receive requests from the client ProcessHacker process. Before a client can make a request to the service, it needs to be verified.

## Kernel Verification of the KProcessHacker Client

Every time an IOCTL is sent to the KProcessHacker service, it is checked to ensure that the caller is a verified KProcessHacker client that is allowed to communicate with the service. All attempts to communicate with KProcessHacker are validated using an IOCTL request key

that is generated by sending a `KPH_RETRIEVEKEY` request from the user-mode process. The importance of this key is discussed in the "KProcessHacker IOCTL Request Keys and APC" section below. Attached to the `KPH_RETRIEVEKEY` request is an Asynchronous Procedure Call (APC) routine, `KphpWithKeyApcRoutine`, which will be executed upon completion.

KProcessHacker initializes a `KPH_CLIENT` structure that needs to be populated with the correct values to distinguish the caller as verified. This initialization occurs when the stager DLL opens a handle to the KProcessHacker driver file. As this occurs, the Windows kernel sends an `IRP_MJ_CREATE` request to the KProcessHacker driver, and the handler, `KphDispatchCreate`, is called.

```
typedef struct _KPH_CLIENT
{
    struct
    {
        ULONG VerificationPerformed : 1;
        ULONG VerificationSucceeded : 1;
        ULONG KeysGenerated : 1;
        ULONG SpareBits : 29;
    };
    FAST_MUTEX StateMutex;
    NTSTATUS VerificationStatus;
    PVOID VerifiedProcess; // EPROCESS (for equality checking only - do not access
contents)
    HANDLE VerifiedProcessId;
    PVOID VerifiedRangeBase;
    SIZE_T VerifiedRangeSize;
    // Level 1 and 2 secret keys
    FAST_MUTEX KeyBackoffMutex;
    KPH_KEY L1Key;
    KPH_KEY L2Key;
} KPH_CLIENT, *PKPH_CLIENT;
```

Figure 5. KPH_CLIENT data structure from the Kph.h source code

This `KphDispatchCreate` function allocates kernel memory to store this data structure. Due to it being kernel memory, the stager DLL is unable to manipulate the data structure from user mode, even from inside the ProcessHacker process. Instead, the stager DLL can send a `KPH_VERIFYCLIENT` IOCTL request to the driver. The handler function, `KphVerifyClient`, for this IOCTL will set the necessary fields once the client is verified.

| IOCTL | Request Name | Description |
|---|---|---|
| `0x99992007` | `KPH_VERIFYCLIENT` | `Verify the client process` |
| `0x999200B` | `KPH_RETRIEVEKEY` | `Retrieve the verification key` |
| `0x999920CB` | `KPH_OPENPROCESS` | `Opens a process` |

| | | |
|---|---|---|
| `0x999920DF` | `KPH_TERMINATEPROCESS` | `Terminate a process` |

Table 3. KProcessHacker IOCTLs used by the stager DLL

The `KphVerifyClient` function first checks to see if verification has already occurred by checking the boolean value, `Client->VerificationPerformed`. If this field is false, the following checks are made by KProcessHacker:

1. Verify that the start address of the APC routine is a user-space address and not a kernel address
2. Compare the process image file name against the mapped PE image name where the APC routine resides
3. Verify that the APC routine address came from an area of memory that is type MEM_IMAGE and in a MEM_COMMIT state
    1. These states ensure that the memory where the APC routine resides is both committed memory and a mapped view of an image section
4. Verify the PE file backing the process making the request by hashing the file's contents and comparing it against a digital signature:
    1. The 256-bit Elliptic Curve Digital Signature is a hash of a known valid ProcessHacker PE file that was signed using KProcessHacker's private key
    2. The digital signature is decoded from the stager DLL and sent with the `KPH_VERIFYCLIENT` request
    3. The contents of the file backing the process that made the request is hashed using SHA-256
    4. The signed hash is decrypted using KProcessHacker's public key
    5. If the decrypted signed hash and the generated hash match, then the PE file is verified

Once verification passes, the code in Figure 6 is executed to populate several fields that will be used for verification when the stager DLL attempts to send the `KPH_OPENPROCESS` and `KPH_TERMINATEPROCESS` IOCTL requests.

```
status = KphVerifyFile(processFileName, Signature, SignatureSize);
if (NT_SUCCESS(status))
{
    Client->VerifiedProcess = PsGetCurrentProcess();
    Client->VerifiedProcessId = PsGetCurrentProcessId();
    Client->VerifiedRangeBase = memoryBasicInfo.BaseAddress;
    Client->VerifiedRangeSize = memoryBasicInfo.RegionSize;
}
Client->VerificationStatus = status;
Client->VerificationSucceeded = NT_SUCCESS(status);
Client->VerificationPerformed = TRUE;
```

Figure 6. KphVerifyClient sets verified fields of a KPH_CLIENT structure

The hijacked ProcessHacker process is now a verified client of the KProcessHacker service. A new thread is spawned to duplicate the IPC objects from DoppelPaymer into the ProcessHacker process space.

### Duplicating the IPC Objects Inside ProcessHacker

From Figure 2, the second argument, `161604546`, is decoded, yielding the handle ID of `0x128`. The section object that this handle references is duplicated in the ProcessHacker process. The section object is duplicated with the same access rights as the original. Duplicating objects generate new handle values, but to keep it simple, this post reuses the original values.

```
//
// Duplicating DoppelPaymer's section object handle.
//
NtDuplicateObject(
     DoppelPaymerProcHandle,  // Process handle for source process
     0x128,                   // Handle for the source section object
     0xFFFFFFFF,              // ProcessHacker process handle
     duplicateHdl,            // New section object handle
     NULL,
     NULL,
     DUPLICATE_SAME_ACCESS
);
```

A view of the duplicated section object is mapped to local process memory using `NtMapViewOfSection`. It contains the same handles from Table 2 that were written to the section object in the DoppelPaymer process: `0x120`, `0x11C` and `0x124`. Each of these handles is duplicated, and a view of the `0x124` section object is mapped into ProcessHacker's process memory.

DoppelPaymer is now in a state where it is waiting for an event to be signaled that notifies it that the stager DLL has completed initialization and is ready to process requests in the queue. This notification is sent by calling `NtSetEvent` with the `0x120` event handle, and the stager DLL waits for requests.

## Killing Blocklisted Applications

Once DoppelPaymer receives the signaled event, it starts enumerating both service and process names, and hashes them with the CRC32 algorithm. These hashes are compared against a list of blocklisted hashes in DoppelPaymer's process memory. The complete list was covered in the previous DoppelPaymer blog post. This section discusses what happens when an application matches one of the blocklisted items.

DoppelPaymer writes the process ID associated with the service, along with a command to the mapped section object, `0x124`. The command will tell the stager DLL which steps to take.

```
AntiAV {
     +0x00   Command
     +0x08   Process ID
     +0x10   errorCodeResponse   // Response code from ProcessHacker
}
```

Figure 7. Blocklisted process information written to IPC section

Process termination occurs in two steps: a process is opened, then it is killed. The first command sent, `1`, will tell the stager DLL to open a handle to the process. Table 4 contains a list of valid commands.

| Command | Action |
|---|---|
| `0` | Terminate ProcessHacker |
| `1` | Open the process |
| `2` | Kill the process |
| Other value | Invalid, wait for the next command |

Table 4. IPC handles with concrete values from testing

The command is written to the queue, along with the process ID, and DoppelPaymer signals the event to notify the stager DLL that data is in the queue. Once that event is signaled, it waits for a response.

## KProcessHacker IOCTL Request Keys and APC

Certain IOCTL requests to the KProcessHacker service require the verification of an IOCTL request key. To ensure that the key cannot be tampered with, the key is generated by the driver and stored in the `KPH_CLIENT` structure. The following IOCTL requests require a key:

- KPH_OPENPROCESS
- KPH_OPENPROCESSTOKEN
- KPH_TERMINATEPROCESS
- KPH_READVIRTUALMEMORY
- KPH_OPENTHREAD

Prior to making any of these requests, ProcessHacker has to send a `KPH_RETRIEVEKEY` request using `NtDeviceIoControlFile`. Along with this request, the user-mode address of an APC routine, `KphpWithKeyApcRoutine`, and the user-mode address of a function called by the APC are sent as parameters. This routine to be called by the APC will end up making one of the IOCTL requests mentioned in the above bulleted list.

```
NtDeviceIoControlFile(
     PhKphHandle,
     NULL,
     KphpWithKeyApcRoutine,  // Called after NtDeviceIoControlFile
                             //  returns
     NULL,
     &context.Iosb,          // Receives the status code
     KPH_RETRIEVEKEY,        // IOCTL
     &input,                 // Parameters passed to IOCTL
     sizeof(input),
     NULL,
     0
);
```

The `KPH_RETRIEVEKEY` request is handled by `KphRetrieveKeyViaApc`. Prior to generating the request key, several checks are performed to ensure that the client (ProcessHacker, in this case) making the request is verified and that the APC parameter is valid:

- Ensure the client has been verified by checking the `KPH_CLIENT->VerificationSucceeded` field
- Ensure that the process information for the client matches what was set during the verification process
  - `KPH_CLIENT->VerifiedProcess`
  - `KPH_CLIENT->VerifiedProcessId`
- Ensure that the instruction address of the APC routine falls within the executable section of the verified client

Once those checks are passed, a request key is generated and stored in the `KPH_CLIENT` structure. This key will also be passed as argument to the APC routine. So now, both the client and the server have independent copies of the same request key. The APC routine, `KphpWithKeyApcRoutine`, executes.

As noted earlier, the APC routine receives a function pointer that will be used to execute a specific action (kill process, open process, etc.). To restrict which requests can be made via this APC routine, it makes sure that only the following functions can be called from the APC:

- `KphpGetL1KeyContinuation`
- `KphpOpenProcessContinuation`
- `KphpOpenProcessTokenContinuation`
- `KphpTerminateProcessContinuation`
- `KphpReadVirtualMemoryUnsafeContinuation`
- `KphpOpenThreadContinuation`

This check prevents DLLs from being injected into ProcessHacker and leveraging the `KphpWithKeyApcRoutine` APC as a method to execute its own routines under the guise of being a valid `KPH_CLIENT`. Once this check passes, the function passed to the APC is

called and the client copy of the request key is passed to that function.

Both this check and the checks made in the `KphRetrieveKeyViaApc` procedure pose a challenge for the stager DLL. The author of the stager DLL scraps the original `KphpWithKeyApcRoutine` routine and passes their own APC routine, `StagerAPCRoutine`. The code for this is written directly after ProcessHacker's overwritten entry-point code. This bypasses both of the function pointer checks and passes the checks performed by `KphRetrieveKeyViaApc`.

```
000000013f5b2f2c    jmp  cs:CallFunctionPointerRoutine
000000013f5b2f3a    push    rax
000000013f5b2f3b    retn
```

Figure 8. StagerAPCRoutine

The `StagerAPCRoutine` (Figure 8) shortcuts the entire process and jumps directly to a procedure, `CallFunctionPointerRoutine`, that calls the DLL stager versions of `KphpOpenProcessContinuation` and `KphpTerminateProcessContinuation` procedures and passes the client copy of the request key as a parameter.

## Opening a Process Handle

As outlined in the previous section, to open a process handle, a `KPH_RETRIEVEKEY` request is sent to the KProcessHacker service. Along with this request, the `StagerAPCRoutine` address and the address of the function called by the APC open a process, `StagerOpenProcess`. A new request key is generated, saved to `KPH_CLIENT` and passed to `StagerAPCRoutine`. Once everything has been validated, the `StagerAPCRoutine` calls `StagerOpenProcess` where a `KPH_OPENPROCESS` request is sent to the KProcessHacker service. Both the client copy of the request key and the process ID of the target are sent with the request.

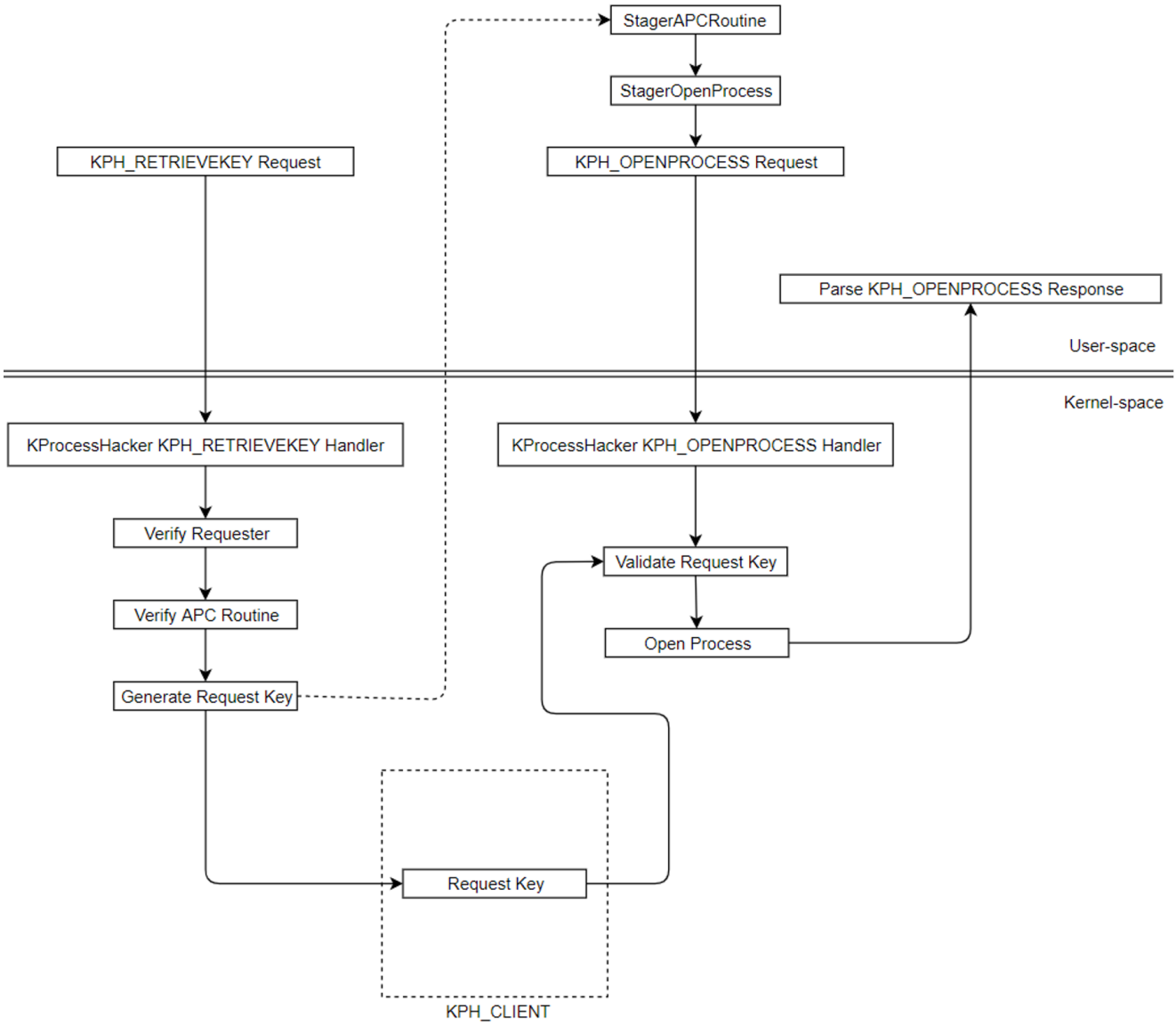


Figure 9. Process diagram to open a process

The service handles this request by calling `KpiOpenProcess`. Before a handle to the process can be opened, the client's request key is validated by calling `KphValidateKey`, where the client copy of the key is compared against the copy stored in `KPH_CLIENT`. If these match, a handle to the target process is opened.

`PsLookupProcessByProcessId` is called to get a pointer to the process object in kernel memory. That pointer is used to open a handle to the object by calling `ObOpenObjectByPointer`. This handle can now be referenced by the stager DLL.

The stager DLL signals the `0x120` event handle, notifying DoppelPaymer that a handle has successfully been opened to the target process. Now the process can be killed.

## Killing a Process

DoppelPaymer verifies that the process was successfully opened, and then takes the appropriate action. If an error occurred, it continues checking for blocklisted applications; otherwise, another notification is sent, this time with the command `2` to terminate the process.

Terminating a process follows the same procedure as opening a process with one difference: The `StagerKillProcess` function pointer is passed to the `StagerAPCRoutine`. The `StagerKillProcess` function sends a `KPH_TERMINATEPROCESS` request to the KProcessHacker service. This is handled by the `KpiTerminateProcess` kernel-mode function. The request key is validated before process termination can occur. The target process is reopened to get a kernel handle, and `ZwTerminateProcess` is called to kill the process. Note that using this procedure ignores PPL, so even protected processes will be killed.

## Conclusion

DoppelPaymer's usage of ProcessHacker to kill AV services is part of a larger trend of various actors leveraging legitimate tools to disable AV/EDR functionality. DoppelPaymer's method is a testament to how innovative malware authors can be when it comes to neutralizing the defenses of their target.

*Many thanks to Bill Demirkapi for helping to sort out how digital signature verification is used to validate the PE.*

### Additional Resources

- *Discover how CrowdStrike Falcon X combines automated analysis with human intelligence, enabling security teams, regardless of size or skill, to get ahead of the attacker's next move.*
- *Find out how to stop adversaries targeting your industry — schedule a free 1:1 intel briefing with a CrowdStrike threat intelligence expert today.*
- *Learn about the powerful, cloud-native CrowdStrike Falcon® platform by visiting the product webpage.*
- *Get a full-featured free trial of CrowdStrike Falcon Prevent™ to see for yourself how true next-gen AV performs against today's most sophisticated threats.*