Evading Userland API Hooking, Again: Novel Attacks and a Principled Defense Method

Cristian Assaiante[0000−0001−7705−0434], Simone Nicchi[0000−0002−0831−8211], Daniele Cono D'Elia[0000−0003−4358−976X], Leonardo Querzoni[0000−0002−8711−4216]

Sapienza, University of Rome
{assaiante,delia,querzoni}@diag.uniroma1.it ; simone.nicchi@gmail.com

Abstract. Monitoring how a program utilizes userland APIs is behind much dependability and security research. To intercept and study their invocations, the established practice targets the prologue of API implementations for inserting hooks. This paper questions the validity of this design for security uses by examining completeness and correctness attacks to it. We first show how evasions that jump across the hook instrumentation are practical and can reach places much deeper than those we currently find in executables in the wild. Next, we propose and demonstrate TOCTTOU attacks that lead monitoring systems to observe false indicators for the argument values that a program uses for API calls. To mitigate both threats, we design a static analysis to identify vantage points for effective hook placement in API code, supporting both reliable call recording and accurate argument extraction. We use this analysis to implement an open-source prototype API monitor, Toxotidae, that we evaluate against adversarial and benign executables for Windows.

1 Introduction

Programmers largely resort to userland Application Programming Interfaces (APIs) to carry heterogeneous tasks such as file and string manipulation, process management, network communication, and many others. For this reason, monitoring what APIs a program invokes represents a common strategy in much dependability and security research to study the externally observable behavior of a program [12]. This practice is also known colloquially as "API hooking" due to the probes (*hooks*) that a monitoring system inserts in the execution runtime to intercept API calls. This paper focuses on Windows programs accessing APIs from userland libraries on x86/x64 architectures, but the general concepts we present in it may be relevant for other platforms, too.

As typically with dynamic analysis endeavors, adversarial programs may attempt to resist monitoring attempts by employing API call obfuscation techniques [19, 12]. Some research has explored defensive methods that increase the robustness of the monitoring through transparent instrumentation [12] or taint tracking [19]. Nonetheless, an enduring weakness that we find in current API monitoring embodiments [31, 27, 28, 12, 16] is that they assume API calls to follow the calling convention of the target platform.

As a result, they record API identity and input arguments once execution reaches an API prologue, and output arguments and return value upon returning to the caller. However, as we show in this paper, both the completeness and the correctness of the tracing are affected if adversarial code invokes APIs in unexpected ways.

Completeness attacks aim to hide calls from monitoring systems. Prior research [19] describes *stolen code* attacks where an adversary emulates instructions from the API prologue and then jumps into API code with a function-boundary unaligned control transfer. While currently known stolen attacks are limited to a few instructions, in this paper we generalize them to much deeper points with modest effort.

Correctness attacks aim to provide fake indicators to monitoring systems and, to the best of our knowledge, are unexplored for userland malware. In this paper, we propose and demonstrate the feasibility of a time-of-check to time-of-use (TOCTTOU) attack on userland Windows APIs, exhibiting benign argument values upon entering the API and replacing them with rogue ones before API code uses them.

To explore the potential extent of both attacks and counter them, we propose a static program analysis technique that tracks how argument values flow through the storage locations in use to API code. Our goal is to intercept call arguments right before API code uses them: this information can guide us in inserting *deep hooks* in a principled way that effectively counters both types of attacks.

We extensively evaluate the attacks and the proposed defense on popular Windows APIs. First, we show that existing solutions are vulnerable to both attacks and that the hooking locations our method identifies are deeper than where either attack can reach. Then, we build an open-source[1] API monitoring system prototype, TOXOTIDAE, using our deep hooks. We test its efficacy against real-world samples that use classic stolen code techniques and against proof-of-concept implementations of our attacks. Following recent literature [12], we extensively test its performance using the Wine test suite: compared to a traditional design, TOXOTIDAE adds an average 7.62% slowdown and recovers API argument values with 99.40% accuracy.

2 Background and Motivation

Windows APIs. Windows offers programmers an extensive collection of APIs through its DLL (Dynamic Link Library) files. A compiled binary can expose a list of APIs it seeks to use and the Windows loader will fill an array, called Import Address Table (IAT), with their run-time addresses. Alternatively, a program may look up an API address at run-time among currently loaded code modules or after loading a new DLL.

Compiled code accesses Windows APIs through well-defined calling conventions for argument passing. On 32-bit systems, all arguments are placed on the stack, typically in 4-byte slots, following the `stdcall` convention. On 64-bit systems, the first four arguments are passed through registers, whereas any following argument is placed on the stack, typically in 8-byte slots. Arguments come with markers (*modifiers*) that identify an input (IN) and/or an output (OUT) value, where an OUT argument is typically a pointer to a storage location for saving a result from the API [12].

Windows API code shows by design several regularities [25]. Functions come with a prologue and an epilogue block. The prologue allocates a fixed-size space[2], which we term **local area**, for hosting local variables and spilling callee-saved registers; on 64-bit API code, the allocation includes also the space needed to call functions internally.

[1] Our artifacts are available at: https://github.com/cristianassaiante/toxotidae.

[2] With the exception of leaf functions [25], which cannot modify the stack pointer and other callee-save registers. Due to their simple nature, they are marginal for this work.

The prologue supplies the area size to an instruction (`sub` or `lea`) that updates the stack pointer accordingly. If the function needs to allocate stack space dynamically (`alloca`), the prologue must set a frame pointer register to mark the base of the fixed part of the stack. The epilogue simply restores the initial value of the stack pointer.

A peculiarity of 64-bit API code is that the caller reserves four stack slots before the return address (and thus the local area). We refer to this space as the **register area**. The called function may use it, if needed, for spilling registers.

Pitfalls of Hooking. To interpose on API calls, current monitoring solutions target for hook insertion either the IAT by replacing its contents or the prologue of API implementations by adding trampolines. The first strategy is an easy prey of adversaries that look up functions at run-time [12]. As for trampolines, adversaries can detect them [12] or, as with attacks to anti-virus userland agents [1], even remove them by restoring the original instructions. Recent research [12] recommends using more reliable and transparent instrumentation mechanisms, such as dynamic binary instrumentation [9] or hypervisor-based invisible breakpoints [22], to mitigate these threats.

However, a natural question is whether intercepting relevant program behavior at system-call level would sidestep the pitfalls of userland API hooking. A common belief is that kernel code is a harder target for attackers to tamper with (but also for defenders to instrument) and that the behaviors of interest will show up in low-level system call activity. When only a coarse-grained characterization of a program is needed, this may be a reasonable compromise: for example, malware sandboxes can instrument system calls to reliably capture in one place behaviors that can be exercised by multiple APIs [11], for example when involving the registry or files.

However, this does not hold for general uses, as well as for other important malware analysis scenarios. One prominent example are anti-virus products: since the introduction of PatchGuard in 2005, they can no longer insert hooks in Windows kernel code, and many rely on userland API hooks as a primary source of information[3]. Another relevant use case is reverse engineering: when attempting, for example, manual analysis of an untrusted object or efforts like malware lineage [17], determining how a program enacts a behavior can be just as valuable as intercepting it in a coarse-grained way.

Semantic loss can also be a problem. For example, network APIs typically see their inputs lowered to a flat data representation for the device driver associated to the connection. This makes, for instance, a `gethostbyname` operation virtually indistinguishable from an `InternetConnectA` one. Other APIs show generic system call activity, or even none: this is the case, e.g., with `CryptBinaryToString`, `GetModuleHandle`, `GetProcAddress`, and other helpers often used by malware authors.

We conclude that userland API hooking remains necessary for general monitoring.

3 Proposed Attacks

This section discusses two attacks to the completeness and correctness of current API hooking systems: an extended form of stolen code and a novel TOCTTOU attack.

As **threat model** and defense capabilities, we make the following assumptions. The adversary knows the Windows build installed on the victim or prepares a payload

[3] At times alongside Event Tracing for Windows (ETW), for which several evasions exist.

for multiple versions. Attacks to the instrumentation technique are out of scope. If attacking one property, the other is not a primary concern for the adversary: for instance, they are not interested in hiding calls to the API undergoing a TOCTTOU attack. Defenses that operate outside userland (e.g., hypervisor-based system call monitoring, ETW-based monitoring) are out of scope and attackers may deploy existing evasion methods if needed: this paper focuses on userland API hooking and its applications.

3.1 Stolen Code Attack to Monitoring Completeness

Kawakoya et al. in [19] report on *stolen code* attacks observed in malware samples in the wild. In such an attack, the adversary copies at run-time[4] some instructions from the prologue of an API to a memory area owned by the sample. Then, when the sample has to call the API, it first executes the copied instructions and then jumps to the instruction in the original API code next to the copied ones.

Stolen code instances reported in literature [8, 29, 21] or that we found in real-world malware and executable protectors (ASProtect, Enigma, Obsidium, PELock, Themida, VMProtect) span the first few instructions (1-3) of an API[5]. Those showcase helpful regularities among the vast majority of APIs. For example, in 32-bit Windows APIs the first instruction is always a `mov edi, edi` that the compiler emits for hot-patching purposes [7]; then, the next two instructions are typically `push ebp; mov ebp, esp`. Thanks to such regularities, implementing an unaligned jump becomes straightforward and already suffices to defeat existing API monitoring solutions.

One could argue that defenders should move their hooks to later instructions. However, two problems arise: how deep a stolen sequence can be and if the API argument values are still visible by then. In this paper, we study how deep such attacks can go with only a modest effort for the attacker. Hence, with longer sequences, one can realize cases where the argument is no longer visible at the original location. Even worse, also branching decisions may become part of the stolen code, potentially requiring monitoring systems to deploy path-sensitive hooks to intercept execution.

Ultimately, an attacker may decide to push the attack to go as deep as possible. We believe a reasonable **boundary** is when the API makes a call. The target can be a helper (typically out-of-scope for hooking [12] as uninteresting) or another API, which the attacker may or may not decide to attack recursively[6]. We remark that stealing an entire function is possible but, in our opinion, hardly profitable. The presence of internal API calls entails a recursive stealing process, with potential dependencies on the Windows version in use and the DLLs involved across calls. As for self-contained APIs, it would be simpler to use an own re-implementation of the functionality.

Instantiating the Attack. We implemented a generalized version of stolen code that targets the call boundary. The workflow is as follows. The attacker prepares a memory region for hosting the API code to steal and locates the run-time address of the API. Then, the attacker copies the entire API body or, more surgically (using

[4] In an attack variant dubbed *sliding call*, the sample contains such instructions already in its compiled body. The distinction is irrelevant for the techniques presented in this paper.

[5] The most complex stolen code attack that we observed in the wild is from the Obsidium protector and encompasses the first 6 instructions of an API.

[6] As advanced monitors filter out internal calls due to their sheer number [12].

a disassembler or precomputed information), just the bytes needed to cover the basic blocks that execution can traverse before reaching an internal call. At each place where a call occurs, the attacker overwrites the opcode with a gadget that, by reading the target from an attacker-configured location, redirects execution as a tail jump to the call instruction in the original API. The adversarial program can call the attacked API with a normal call to where the stolen code resides, and pass the parameters according to the calling convention.

As we show in Section 5, our extended attack enables the stealing of dozens of instructions. At such depths, API arguments may no longer be visible at the original locations. Furthermore, a deferred single hook may not suffice to intercept API execution due to argument-dependent control-flow choices taken in the stolen code. For example, on the DLL collection that we study in Section 5, our attack copies path-sensitive code for about 53.77% of the APIs in Windows 10 64-bit.

3.2 TOCTTOU Attack to Monitoring Correctness

Time-to-check to time-to-use (TOCTTOU) race conditions create a window of opportunity for an attacker when a data item is accessed a first time for checking purposes and then a second time for modifying execution state. During this window, the attacker may deliberately alter the data and subvert the semantics of the initial check. TOCTTOU attacks are well documented in UNIX-like systems, notoriously affecting file systems [33] and the userland/kernel interface [4].

In our context, we propose and demonstrate the feasibility of a TOCTTOU attack on userland API monitoring, targeting the window between call argument inspection at the API prologue and when API code later acts according to such values.

Main Idea. As discussed before, current systems retrieve API argument values right before executing the API prologue, inspecting the appropriate registers and stack locations according to the calling convention and the API prototype. To hamper monitoring correctness, an attacker may invoke the API with misleading (e.g., innocent-looking) argument values to deceive the logging or security policy in place and eventually replace them with the intended ones.

A vulnerable time window opens up between the value logging and the first use of the argument value in API code, where the use is identified after excluding trivial copies (e.g., for register spilling). From a separate thread, the attacker targets the location from which the first use (and subsequent ones, if any) takes place and updates the value with the intended one. As we will see, due to internal data movements, such location often differs from the one where the API initially received the value.

In the following, we assume to have an oracle for determining the visibility and location of API argument values at different points in API code and where their first uses occur. Section 4 will present program analysis techniques supporting these tasks.

Vulnerable Locations. As the attacker issues the API call, the stack pointer value upon entering the API is known. The attacker can use a relative displacement to write the desired value to the location from which API code reads the value for its first use.

The attack we propose is not limited to the argument locations dictated by the calling convention. It also benefits from data movements that API code makes when copying an argument to a spilling slot or to a local variable that will hold it for the

remainder of the computation. The oracle informs the attacker on the location of such copies and where the first "real" use of an argument value happens, either as an operand of a non data-movement instruction or if passed as argument to an internal call.

We identify three types of potentially vulnerable locations:

❶ *On-entry slot.* The attacker overwrites the stack location where the caller passes the argument (Section 2). A shortcoming of this case is that, on 64-bit code, the first four arguments of any API are out of reach as they reside in registers.

❷ *Local-area slot.* The attacker overwrites a copy of the argument value that API code makes to a location in the *local area* (Section 2) of its stack frame. This happens, for example, for register spilling purposes. Eventually, the program accesses this copy to read the argument value and make use of it.

❸ *Register-area slot.* This case is analogous to the previous one, with the copy being located in one of the slots of the *register area* (Section 2) that the caller reserves when invoking a 64-bit API. The API code may spill there a copy of an argument (typically one passed via register, but not necessarily so) and eventually re-read it.

Figure 1a shows the distribution of such vulnerable locations across the APIs available in popular DLLs on selected versions of Microsoft Windows. While ❶ is the most frequent case, on the 64-bit build types ❷ and ❸ account for the majority of locations.

Timing the Attack. For a successful attack, the attacker should overwrite a target memory location between the check (i.e., when the API hooking system logs argument values) and the first use from API code. As we will measure in Section 5, this window typically encompasses a dozen or more instructions, and in our experiments we were able to attack windows of even just two instructions with good success.

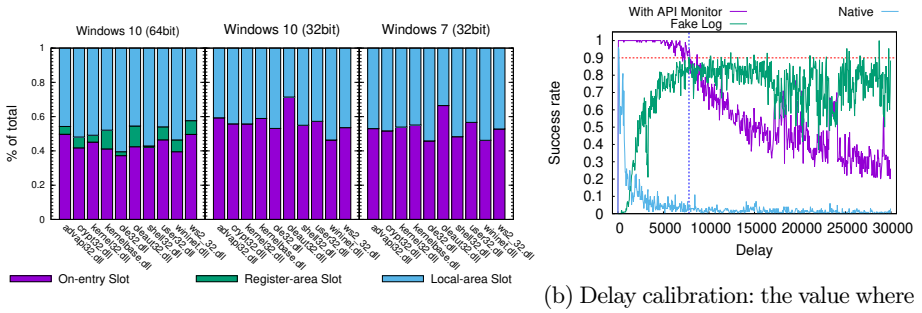
As the attacker controls when the call is issued, locations of type ❷ or ❸ can be attacked in a straightforward and reliable way. As they are not monitored, an attacker may set a loop that overwrites the location even before the API call takes place[7].

For locations of type ❶, the difficulty for a successful TOCTTOU attack is to time accurately when to start the overwriting. This should happen only after the API monitoring system has logged the initial fake value (or it would log the rogue one) and before its first use. Furthermore, the latency of the API call processing is system-dependent and, in the general case, unknown. For this task, we envision two avenues.

For high reliability, the attacker may build a deterministic method to intercept when execution reaches a certain API instruction and then overwrite the location. As one possible embodiment, the attacker sets an exception handler and then a hardware breakpoint at the desired API instruction. We find this a safe evasion option when the goal is to impede reverse engineering (Section 2), but it may be realistic also against security solutions, as shown by recent attacks that disable security products [18] or the Windows Anti-Malware Scanning Interface (AMSI) using hardware breakpoints. Less refined variants, for example based on software breakpoints, are also possible.

Alternatively, the attacker may be willing to accept occasional failures in exchange for simpler deployment and maximum stealth. For this scenario, we came up with three techniques to determine with high probability that the monitoring has ended.

[7] The attentive reader may argue that this holds only if a slot is not reused with different variables: in this case, the overwriting should be timed (e.g., alike to what we do for ❶) or correctness may be affected. However, in our tests, slot reuse appeared quite rarely.



(a) Type distribution for vulnerable locations

(b) Delay calibration: the value where the two lines intersect is a good pick.

Fig. 1: Breakdown of TOCTTOU-attackable locations and feasibility in hardest setting.

The first two are surgical and mainly reliable: to time the overwriting, they wait for some sentinel value to appear on the stack as part of API code execution. The third is more general and attempts multiple TOCTTOU attacks with a benign value on an API to indirectly measure the delay introduced by the monitoring.

Sentinels. One sentinel kind are *stack immediates*: i.e., immediate values that API code pushes to fixed stack locations. A typical example in 32-bit API code is the size pushed on stack as argument when calling internally `__SEH_prolog()` to configure Structured Exception Handling. The attacker implements a busy-waiting loop that inspects the stack memory until the known constant appears: when it does, execution has left the monitor, and the attacker can start overwriting the target location.

Another sentinel kind are *spilled registers*: i.e., registers that API code spills to fixed stack locations and whose content is known to the attacker. The typical case are callee-saved registers, which the attacker can arbitrarily set before calling the API, but also API argument values and even the base pointer value (from the `push ebp` in the API prologue) can serve the purpose in a busy loop designed as above.

Delay Loop. When no sentinel is available, the attacker may exploit TOCTTOU race conditions also to guess when the monitoring is over. The attacker chooses two non-suspicious values for an API of interest, calls it with one, and starts the overwriting operation with the other with a delay. Then, the attacker inspect the result of the API (or its effects on the system) to determine which value was used.

The attacker employs a *delay calibration loop* to gradually vary the delay and measure the stability of the result on multiple calls. The goal is to find a delay sufficiently high such that, if increased further, the overwriting will likely happen after the use; aiming for a high delay provides the attacker with confidence that the monitoring has, by then, ended. From then on, to call the API with conspicuous arguments, the attacker issues only a single call and adds the measured delay to the thread carrying the overwriting presented for the main attack. The value can be reused to attack other APIs with a similar expected logging time (e.g., for argument count and types).

Figure 1b plots the effects of the delay, measured as the trip count of a `loop` assembly instruction for busy-waiting, in a native execution and under a popular [12] hooking tool, *API Monitor*. The attack targets the `WSAStartup` API. In the Windows

10 build used for the test, the API offers a TOCTTOU window size of 21 instructions. The attacker identifies a high delay value that results in the API executing with the rogue argument with, at least, the desired probability. As we inspected the logs of this and similar experiments, we saw that a very high success rate in timely argument overwriting is associated with the API monitoring system logging the fake initial values. In the figure, the point where the dashed lines insersect shows a 91% success rate if the attacker waits for the corresponding delay before starting the attack.

Discussion. The TOCTTOU attack we propose is the first targeting API hooking correctness and, to the best of our knowledge, also the first entirely within userland. We target a design weakness of current hooking systems, exposing fake indicators for argument values and then overwrite them as late as possible, either in their original location or in later copies depending on the opportunities that the victim API offers. We remark that the attack applies also to OUT arguments, as the attacker can put a fake result in the original buffer and have the API write the result to a new buffer.

Our goal is to show that TOCTTOU attacks are feasible. We note that many targets (type ❷ and ❸) are straightforward and we propose alternatives to comprehensively treat locations of type ❶ with different degrees of reliability and complexity.

The next section presents techniques that examine API code for identifying what are the instances of an argument value and where the code uses them. While we will use them for defensive purposes (including the defeat of our attack), they can also measure to what extent Windows APIs may be vulnerable to TOCTTOU races.

An orthogonal aspect worth mentioning is that the delay loop technique may be used to detect if a hooking system is present. In particular, we noted that the distribution of the delay value compared to its effects on the API outcome changes significantly between native and monitored executions, with the latter needing much higher values.

4 Towards a Principled Defense

This section proposes countermeasures for more reliable API hooking designs. In the following, we present program analysis techniques to track the propagation of the values provided as call arguments throughout API code and study their liveness, which is important in turn in order to identify what value copies may be accessed.

4.1 Overview

A clear weakness of current API monitoring systems is their on-entry hooking policy. We argue that placing probes in deeper spots would significantly reduce, if not close, the attack surface for completeness and correctness attacks as those from Section 3.

To this end, the main obstacle to overcome is that, for deeper spots, defenders should determine until when argument values are still visible, including in the reasoning the locations where they may have flown through data movements. For example, compilers perform a register allocation process to control which intermediate values and expressions reside in registers, and when a register must be temporarily freed for other computations, the compiler spills its value on the stack [6]. Also, API code may read argument values and host them in temporary, internal variables before using them. Even debug information available for Windows DLLs as PDB files is insufficient to track these movements. Therefore, we designed a binary-level static analysis to track the propagation of API call argument values through the control-flow graph (CFG).

Main idea. We want to determine the deepest point(s) in the CFG of the API where one can place hooks and still be able to determine not only that an API call took place, but also which were the parameters supplied for it. We aim for a data-flow analysis that can trace parameter value propagation from the entry point of the API. More precisely, we want to see how the values they contain flow through API code and where they end up stored (in spilling slots or copied to internal variables hosted on stack or in registers) for eventually being read again for computations that make use the value (e.g., branches, arithmetic manipulations, supplying an argument to an internal call).

The most convenient place where a parameter can be retrieved, either in its original location or through a copy, is where API code uses its value last. This scenario covers both unaligned invocations (as with stolen code schemes) and TOCTTOU attacks[8].

The data-flow analysis we design computes for every basic block in the API an input list describing which parameters are visible and their current location(s) when execution enters the block. Then, we evaluate how instructions in the block can affect such list (for instance, by polluting a register containing a dead value), and produce an output list for the block. For any block, the input list is given by a merge operation where we intersect the output lists of the blocks with an outgoing edge to it.

The input and output lists that we build for the data-flow analysis also allow us, through further refinements, to precisely locate when the API code accesses a parameter value, regardless of its availability for inspection. Identifying where uses happen is important because an attacker may apply the TOCTTOU idea to overwrite the location after the use and mislead a hooking system that inspects a value "too late".

Finally, relying on topological properties of the CFG for efficient insertion, we identify suitable places for placing hooks to log each API parameter, so as to cover all the alternative execution paths in the CFG while being at the deepest possible places.

4.2 Tracking Parameter Value Propagation

We model the control-flow graph as a directed graph $G=(V, E)$ with a set of vertices V and edges E. The API prologue corresponds to vertex v_0. We assume there is a single vertex with no incoming edges and that there are no isolated vertices.

Algorithm 1 is designed to work on compiler-generated code that is well-behaved [23] in terms of stack manipulations: by that, we mean that every basic block can alter the stack pointer register only by a predetermined quantity and that CFG join points (i.e., at basic blocks with multiple predecessors) see an identical stack pointer variation across all the incoming paths compared when execution reached the prologue v_0. We note that Windows API code naturally meets these characteristics (Section 2).

For each vertex $v \in V$, the algorithm maintains two sets $PAR_{in}[v]$ and $PAR_{out}[v]$ that it updates using a fixed-point approach. Initially, $PAR_{in}[v]$ and $PAR_{out}[v]$ are not instantiated (we use $\perp$), with the exception of $PAR_{in}[v_0]$ that we populate with the set of pairs for each argument and its location at the prologue according to the calling convention. The algorithm uses a worklist W with initially $W=\langle v_0 \rangle$.

The algorithm pops a vertex v from the worklist, and if $PAR_{in}[v]$ is not initialized, it computes it by merging $PAR_{out}[v_{pred}]$ for each predecessor v_{pred} of v. Informally, all the results from the computation of the predecessors of vertex v are merged

[8] In case of multiple uses, the attacker has to overwrite the value already before the first use.

Algorithm 1 Proposed Algorithm for Tracking Parameter Value Propagation

1: $\forall v \in V - \{v_0\} : PAR_{in}[v] = \bot$; $\forall v \in V : PAR_{out}[v] = \bot$
2: $PAR_{in}[v_0] = \{(arg_i, loc_i) \mid$ value of arg_i is at loc_i by calling convention$\}$
3: $W = \langle v_0 \rangle$
4: **while not** $W.empty()$ **do**
5: $\quad v = W.pop()$
6: $\quad I = \{PAR_{out}[v_{pred}] \mid (v_{pred}, v) \in E \wedge PAR_{out}[v_{pred}] \neq \bot\}$
7: $\quad$ **if** $I \neq \emptyset$ **then**
8: $\quad\quad new\_in = \text{merge}(I)$
9: $\quad\quad$ **if** $PAR_{in}[v] == new\_in$ **then** *continue*
10: $\quad\quad PAR_{in}[v] = new\_in$
11: $\quad new\_out = \text{compute}(v, PAR_{in}[\text{v}])$
12: $\quad$ **if** $PAR_{out}[v] \neq new\_out$ **then**
13: $\quad\quad PAR_{out}[v] = new\_out$
14: $\quad\quad$ **for each** v_{succ} *s.t.* $(v, v_{succ}) \in E$ **do**
15: $\quad\quad\quad W.push(v_{succ})$

together. After the initialization of $PAR_{in}[v]$, the algorithm computes statically the effects on memory and registers of the instructions in the basic block and computes the output state. Mainly, the static analysis recognizes the introduction of value copies to new locations through data movement instructions, while it discards a currently known location when its contents are overwritten. If the output state differs from prior iterations, $PAR_{out}[v]$ is updated and all the successors of v enter the worklist.

We treat information from multiple predecessor blocks via set intersection. In particular, the merge() operator takes all the pairs (arg_i,loc_i) such that $(arg_i,loc_i) \in PAR_{out}[v_{pred}]$ for all the predecessors v_{pred} of the current vertex v. Informally, an argument-location pair is added to the result set iff. it is in the PAR_{out} set of each predecessor. This approach is the most conservative possible for value correctness, as it discards information for locations whose visibility is potentially path-sensitive.

For the static analysis of compute(), we track registers, memory contents, and related aliases; we include provisions for more complex cases of pointer expressions on structure fields and nested dereferencing. For instructions that modify operands for purposes other than data movements, we clear from the set all the arguments that see their location altered by the instruction. This choice is conservative too, as compilers may realize data movements also through arithmetic/logic instructions: however, in our experience, such cases are rare in Windows API code; therefore, we opted not to include heuristics for them, avoid the risk of introducing unnecessary imprecision.

4.3 Identifying Live Copies of Argument Values

The conservative approach taken at merging data-flows in Algorithm 1 is adequate for retrieving call argument value against traditional API obfuscation attacks. That is, when the analysis reports a location for a parameter value at a certain program point from API code, the value will be identical to the one visible upon entering the prologue. However, if the adversary attempts TOCTTOU attacks like the one we described in Section 3.2, this property no longer holds.

Consider a scenario, rather frequent in our observations, where API code maintains two copies of an argument value. At some point, API code no longer uses the first copy, and does not reuse its storage location: therefore, the copy remains visible at later basic blocks, ideally even until the API epilogue. From that point forward, API code accesses instead the other copy and, for example, issues an internal API call using it or makes other program state changes. If the storage location of this other copy is vulnerable to a TOCTTOU attack, an API monitoring system that logs only the first copy (for example, just because it is visible longer) will see only the initial parameter value, missing the value that the adversary sets instead via the attack.

Therefore, at every program point we should look for what parameter value instance(s) API code may access and use in the remainder of the execution. This problem is similar to what in compiler theory goes by live variable (or liveness) analysis [26]. In particular, a variable is said to be live at a program point if the program may access it in the remainder of the computation without reassigning it first. Alternatively, this can also be modeled as a reaching definition problem [26]. Techniques for conducting either analysis on source code are well established.

To cope with our scenario, we first extend Algorithm 1 to compute PAR_{in} and PAR_{out} sets at the granularity of individual instructions. Then, we adapt the function compute() to construct the standard $LIVE_{in}$ and $LIVE_{out}$ sets of a standard backward liveness analysis, again at the granularity of individual instructions. Therefore, $LIVE_{out}[i]$ contains the locations that the program may reference in the reminder of the execution past i (i.e., the locations that are live after i), and $LIVE_{in}[i]$ contains the locations that are live at i. At control-flow join points, the backward analysis assigns $LIVE_{out}[i]$ as the union of the sets $LIVE_{in}[j]$ where j is an immediate successor of i.

Therefore, to solve our problem, we can intersect the locations from $LIVE_{in}[i]$ with the locations that appears in the pairs at $PAR_{in}[i]$ as hosting a parameter value. This will remove stale copies of parameters values from the results of Algorithm 1, exposing the sole instance(s) that a monitoring system should consider as valid.

4.4 Hook Placement

Knowing which copies of parameter values are live at each instruction in API code provides sufficient information for an API monitoring implementation to reliably trace API call parameter values. The last element of our approach to counter completeness and correctness as in Section 3 is to determine what are the most convenient places for hook insertion to combine efficacy (i.e., to defeat the attacks) with performance.

In general, aiming for the deepest places is a good strategy as long as all the alternative execution paths throughout the API traverse them. Unfortunately, not all paths are similarly deep. This implies that some paths will need an early logging, whereas for deeper paths we may have to re-inspect the value against TOCTTOU attacks.

We can address this problem by building dominator information on the CFG and using it as follows. For each leaf node in the dominator tree and for each API parameter, we see if a live copy of the parameter value is visible at any of its instructions, starting from the last in the basic block. If found, we mark the instruction as a place for hook insertion. Otherwise, we inspect its ancestors until a copy becomes available and we hook it. The intuition behind the criterion is that we want to defer logging values unless their visibility becomes path-sensitive.

4.5 Implementation

We implement the analysis techniques from the previous sections in Python in about 2200 LOC. For disassembly and CFG reconstruction, we use off-the-shelf tools: in particular, we opt for IDA Pro [13] as it also natively integrates with the PDB debug information files that Microsoft makes available for Windows DLLs. These files report the stack pointer variation at each instruction compared to its value at the prologue. We retrieve this information directly from IDA, including the variations induced by internal function calls when they clean up the stack upon return. For API prototypes, we resort to existing efforts [12] based on automatic extraction from Windows header files. Finally, to capture the semantics of instructions, we use the Capstone [5] framework to recover operands (including the individual components of complex memory addressing expressions) and memory access kinds (read, write, or both).

We then implement TOXOTIDAE, a prototype API tracer based on the design we argue for in this paper. For its realization, similarly as in the most recent work in the area [12], we use Pin [24] for dynamic binary instrumentation. As we only need to hook specific addresses, we remark that also implementations based on virtualization-assisted hardware breakpoints or even user-space inline hooks remain possible. We provide TOXOTIDAE with precomputed program points for hook insertion (currently for selected Windows 7, 10, and 11 builds, but adding more is straightforward) and with prototypes for all the APIs from the DLLs of interest for monitoring.

4.6 Discussion

The program analysis techniques we presented allow us to study the propagation and life span of parameter values throughout the locations in use to a function. We use this information to revisit the canonical design of API hooking solutions, identifying multiple spots for hook insertion that account for the plurality of alternative paths and the decaying visibility of value copies. Each parameter sees the insertion of dedicated hooks; as we measure in Section 5, their number is small in practice.

The output of Algorithm 1 also provides an obvious way for computing vulnerable locations for TOCTTOU attacks according to the patterns ❶-❷-❸ we showed in Section 3.2. In Section 5, we will experimentally measure that our defensive hook are deeper than the reach of both our extended stolen code attack and the TOCTTOU one.

Finally, our analysis techniques may be of general interest. For example, we believe they may be helpful to amend debug information that optimizers discard inadvertently [2] or emit incorrectly [10] due to inaccurate implementations, by comparing it with the storage locations we track. We leave this investigation to future work.

5 Evaluation

For evaluating the attacks and the defense methodology presented in the paper, we seek to answer the following research questions:

RQ1: How are common Windows APIs vulnerable to our attacks?
RQ2: How many hooks are needed for the proposed defense? How deep are they?
RQ3: What are the accuracy and performance costs of the proposed defense?

We conducted our evaluation mainly on two virtual machines running Windows 7 SP1 32-bit (build 7601) and Windows 10 64-bit (version 1803). The host machine

Table 1: DLLs and analyzed APIs (H: Hooks, P: Parameters, A: API).

DLL	Windows 10 (64bit)				Windows 10 (32bit)				Windows 7 (32bit)			
	Analyzed	Parametric	H/P	H/A	Analyzed	Parametric	H/P	H/A	Analyzed	Parametric	H/P	H/A
advapi32.dll	626	612	1.092	3.6127	643	632	1.0513	3.6345	680	669	1.1936	4.142
crypt32.dll	258	258	2.8088	10.6279	284	280	2.464	9.5643	284	280	2.1117	7.8143
kernel32.dll	1152	1147	1.076	2.476	1158	1150	1.0477	2.6296	1109	1097	1.3872	3.6737
kernelbase.dll	1315	1309	2.1804	5.8892	1352	1347	2.0753	6.0371	504	502	2.0277	5.9801
ole32.dll	239	228	2.2024	5.7061	218	211	2.1315	5.6161	315	305	2.3748	5.9607
oleaut32.dll	407	406	3.024	7.8818	395	395	2.7983	7.3671	392	392	2.4534	6.5434
shell32.dll	330	300	1.6542	4.3633	379	354	1.5549	4.1695	404	392	1.728	4.8673
user32.dll	679	658	1.7289	4.114	676	670	1.5849	4.0	672	665	1.3846	3.588
wininet.dll	275	264	2.8932	10.4735	298	276	2.3741	8.5688	295	274	2.6854	9.6131
ws2_32.dll	161	161	2.7717	9.8944	149	149	2.9597	10.1946	147	147	2.288	7.585

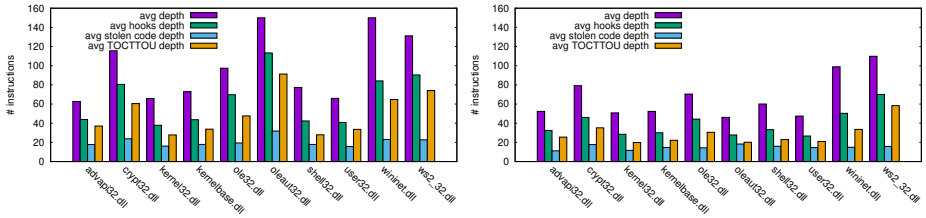


Fig. 2: Average depths. Left: Windows 10 (64-bit). Right: Windows 10 (32-bit).

has an Intel Core i7-11800H CPU @ 2.30GHz with 8 physical cores (16 threads) and 32 GB of RAM. Each VM received 4 virtual CPU cores.

Table 1 shows the 10 DLLs we select for the study, following the choices of the authors of the SNIPER tracer [12]. These DLLs feature heteregeneous APIs used both in malware and goodware, covering disparate tasks from file manipulation to network communication and cryptography.

We analyze only APIs that take arguments. Then, we further check if they have at least one argument that is live at the prologue, labeling them *parametric*. APIs not in this group are deprecated or dismissed APIs or return constant values.

For comparison, we consider three popular monitoring tools (Rohitab API Monitor, SpyStudio, WinAPIOverride) and a state-of-the-art academic system (SNIPER [12]).

Attacks. To answer **RQ1**, we first study in Figure 2 how deep that the extended stolen code and the TOCTTOU attack can go. For a meaningful comparison, we consider for the chart only the functions that are amenable to both. Provided as a reference, the average function depth of functions is measured as the length of the maximal acyclic path in the CFG in terms of traversed instructions.

We recall that TOCTTOU attacks on 32-bit APIs are always possible thanks to pattern ❶, hence the distinction is relevant only for 64-bit API. For the other attack, to keep the analysis realistic, we consider only functions with at least 10 instructions and that are not stubs that jump into another API (or an attacker would just attack it).

The data for stolen code on the 64-bit APIs not amenable to TOCTTOU attacks do not show relevant differences, hence we omit them for brevity. For similar reasons, we omit details on the 32-bits DLLs of Windows 7 and refer for them to the Windows 10 counterparts. The average function depth differs between the 32-bit and 64-bit APIs mainly because vulnerable 32-bit APIs are a superset of vulnerable 64-bit APIs.

DLL	Vulnerable APIs (%)	Redirecting	Parametric
advapi32.dll	147 (49.00)	312	612
crypt32.dll	122 (51.19)	23	258
kernel32.dll	128 (32.82)	757	1147
kernelbase.dll	432 (36.12)	113	1308
ole32.dll	40 (24.69)	66	228
oleaut32.dll	41 (11.61)	53	406
shell32.dll	55 (26.83)	95	300
user32.dll	127 (32.15)	263	657
wininet.dll	104 (40.94)	10	264
ws2_32.dll	67 (43.23)	6	159

Fig. 3: APIs vulnerable to TOCTTOU attacks in Windows 10 64-bit APIs.

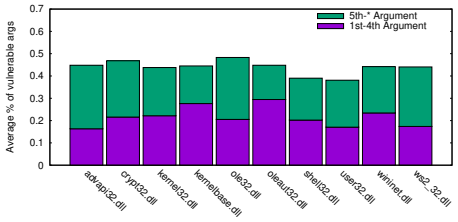


Fig. 4: Arguments vulnerable to TOCTTOU attacks in Windows 10 64-bit APIs.

Extended Stolen Code Attack. We can immediately see from Figure 2 that depths of about 1–2 dozens of instructions are possible. Figure 6 (Appendix A) provides additional data points, including the median absolute deviation. These depths are significantly larger than in currently documented stolen code instances (Section 3.1).

Also, they back our claim that current on-entry hooking designs cannot be amended by simply pushing their analysis a few instructions deeper. Not only argument locations may change (especially with 64-bit code), but further analyses we carried on these Windows 10 64-bit APIs showed that about 53.77% of the code sequences our method can steal include path-sensitive code (which requires multiple hooks for interposition).

To show the feasibility of the method, we implemented a script (Section 3.1) that automatically and surgically steals API code until the call boundary. We use it to shield synthetic test programs, as well as several calls in the code of Al-Khaser, a popular utility for stressing malware analysis environments. For the latter, we test 21 APIs (such as `AdjustTokenPrivileges`, `CheckRemoteDebuggerPresent`, `EnumProcesses`, `SleepEx`, and `VirtualProtect`) with heterogeneous depths. All protected calls operate correctly both in a native execution and when under the four monitoring systems we chose for the evaluation, going unnoticed by all of them.

TOCTTOU Attack. The average depth of vulnerable locations reported in Figure 2 represents the average size, in terms of instructions, of the TOCTTOU window that an attacker can target before API code uses a parameter value (for purposes other than making a local copy). On 64-bit code, these windows are deeper: this is due to the fact that, as we discuss next, fewer arguments see their values being stored in vulnerable locations—whereas all APIs and all arguments are attackable on 32-bit DLLs.

Figure 3 shows how many 64-bit parametric APIs from Table 1 are vulnerable. We further refine parametric APIs by discarding redirection stubs (similarly as with stolen code), as for an attacker it would be more convenient to attack their targets directly. Therefore, we report the number of APIs that we can attack in one or more argument and express them also in terms of percentage of non-redirecting parametric APIs.

Figure 4 shows how frequently arguments are attackable: the total amount is computed as the distinct vulnerable arguments among those actually used by each vulnerable API inside the DLL. Then, we further distinguish the identity of arguments between those passed by registers (from the first up to the fourth) and those passed via memory. Overall, across all vulnerable APIs, we may attack nearly half of their arguments.

Finally, regarding vulnerable storage locations, we already reported the distribution of locations types ❶-❷-❸ in Figure 1 when presenting the attack in Section 3.2.

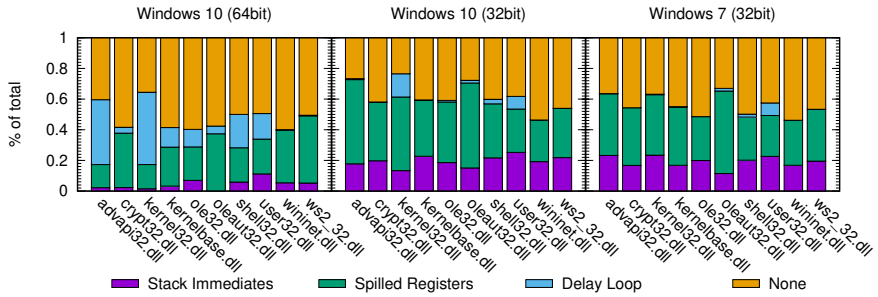


Fig. 5: Deferred overwriting strategy needed at TOCTTOU-vulnerable locations.

Here, to complete our investigation on the potential attack surface of Windows APIs, we provide additional insights on how we can attack locations of type ❶. Figure 5 analyzes all vulnerable locations and shows as "none" those that we classify as ❷ or ❸ (which are straightforward to attack, Section 3.2). Then, it divides locations of type ❶ according to the method needed to attack them. To recap, either the attacker may benefit from sentinel values or they have to explicitly determine (with a delay calibration loop or with a breakpoint-like mechanism) when the on-entry monitoring is over. We remark that the latter is the hardest but also, by far, the least frequent case in practice.

Summarizing, TOCTTOU attacks are possible and require sophistication in the overwriting step only for a limited fraction of parameters. 32-bit APIs are naturally more vulnerable due to stack-passing for all arguments, whereas for 64-bit (with the exception of the APIs from `oleaut32.dll`) the DLLs in our study show that 11.61 to 51.19% of non-redirecting APIs are vulnerable in one or more of their arguments.

As for feasibility, we conducted a preliminary investigation with several proof-of-concept implementations (PoCs) on 12 machines featuring different CPU models (9 Intel Core models from 5 generations and 3 AMD Ryzen from 2 generations) and Windows versions (Windows 11 Home builds 22631 and 22621, Windows 10 Home 19045, Windows 10 Education 17134). The PoCs cover multiple APIs (e.g., `CryptBinaryToStringA`, `ShellExecuteA`, `RegOpenKeyA`, `WSAStartup`, `inet_ntop`) and all the location types and overwriting strategies presented in the paper. Each PoC attempts 20 executions of the API and checks whether it executed using the initial or the overwritten parameter(s). Across our tests, we obtained a success rate of 97.06%. When inspecting the failures, we noticed that they originated from a fragile monitoring choice on one stack-immediate sentinel value, which was selected too close to the instruction accessing the parameter location under attack.

After excluding this program, we repeated these tests on the machine used for the main evaluation using the four API monitoring tools chosen for the study. None of the tools was able to log any rogue argument value in the 20 trials.

However, while the others operated correctly, API Monitor resulted in several crashes on 64-bit code. We noticed that it alters the stack layout significantly: its hooking stub uses the register area that the program sets up for the API (Section 2) to spill register-passed arguments, copies stack-passed arguments lower in the stack, and restores register-passed arguments before issuing the call to the API. This has two drawbacks: transparency, as its hooking is straightforward to detect by monitoring

the original register area contents, and security, as now every arguments becomes a type ❷ location and we can attack it reliably by just adding a predetermined offset.

Defenses. Table 1 and Figure 2 contain relevant information for discussing the effectiveness of the proposed defense (**RQ2**). The key insight from Figure 2 is that the hooks we can place thanks to our value tracking analysis are appreciably deeper than the reach of both extended stolen code and TOCTTOU attacks.

Besides average values, we checked that no attack path is deeper than where we put our deepest hooks. When this hold, the hooks we insert counter completeness and correctness attacks by construction. Across all the APIs we tested, the only failure was one stolen code case surpassing the last point where our analysis deems an argument visible (hence we hook it). A bug in the analysis had made us miss a copy available within a struct, despite the analysis tracked the storage of the struct correctly.

The path-sensitive insertion policy guided by dominator information that we use for hook insertion results in a number of hooks that is rather limited on average. In Table 1, we plot as H/P the average number of hooks that we need for each parameter of the API throughout the CFG. The average is computed on a per-API basis and then aggregated as a geometric mean across APIs. Across the three DLL collections we study, we find that this number is in the range $1.0477-3.024$. Then, we plot as H/A the average number of hooks that we insert for each API. This number is in the range $2.476-10.4735$. While H/P reflects the complexity of the interactions in API code over argument values, H/A reflects the opportunity that we take for optimizing hook insertion, grouping per-parameter hooks when at identical program points. Furthermore, in practice, almost the entirety of these hooks execute only once per API call, as for the DLLs that we study they are inserted within loops only rarely.

Testing. To test the efficacy of our TOXOTIDAE system, we successfully run the PoCs that we synthesized for the extended stolen code and TOCTTOU attacks. We also test real-world instances of traditional stolen code attacks, in particular executables shielded with protectors (Obsidium, Themida, VMProtect) and 10 malware samples (Table 2). We obtained the samples thanks to the authors of API Chaser [19], who shared with us the 2 samples used for its evaluation, and to a professional malware analyst that spotted them during their daily job; the 8 samples are mainly from 2020 and 2021. As expected, all these executables successfully evade the four existing API hooking tools we tested, whereas TOXOTIDAE traces their calls correctly.

An interesting sample is `051...dc4`, which invokes the `LoadLibraryA` API normally excepts when it loads, using stolen code, `cmcfg32.dll` for configuring the Microsoft Connection Manager for remote accesses. We also find it interesting to report on `08c...4b1`, a sample presumably protected with Obsidium. After invoking in a normal way typical APIs for code unpacking, it performs evasive sequences around debugger and virtualization artifacts (e.g., `CreateFileW` on `\??\VBOXGUEST`) using stolen code. We believe this is done to frustrate analysts that try to analyze the sample using manual debugging tools that do not automatically counter these tactics.

Performance. Finally, we tested accuracy and overheads of TOXOTIDAE using the test suite of the Wine emulator for the DLLs selected for our study. This suite, used also in the evaluation of SNIPER [12], stresses different implementation aspects of APIs; also,

Table 2: Analyzed malware samples. `5fd..` and `e09..` are from API Chaser [19].

Hashes (md5)	
9407345f8a1d891624fcba99d20a8b33	61f5e1d9bdc23b0328ef7e874314e3a7
886a19aa14a41386dfb695ad908a1999	08cfaf120cd12bafe53c2fbb50c204b1
33fb8185801d229c91d4aef1efba941d	de71c06b30caa7488f6f42380316ef18
f1cfdb7530169e9398bcc78095ecdcf3	0519f94bf975fdc6b92b40f282853dc4
5fd727d3c11c66583f92970e4c0ec197	e0974042a67ad3db9042e16e4dcb0465

being deterministic and more API call-intensive than real-world applications, it enables reliable accuracy tests and simulates worst-case overhead scenarios. While the authors of SNIPER picked one workload per DLL, we used all the 130 available for our DLLs.

For accuracy, we configure TOXOTIDAE to log parameters also upon entering an API and systematically compare their values against those from our deeper hooks. Across all tests, our hooks retrieve correct values in 99.40% of all invocations of 64-bit APIs and 99.82% on 32-bit APIs. We find this result encouraging for a static analysis.

The main cause of inconsistency in our results are partial uses of argument locations that our static analysis does not model correctly: for example, a `BOOL` type on Windows is sized as an `int` but most APIs only use its least significant byte.

For hook coverage, we compute the geometric mean of the distinct hooks hit across all tests among those we set for an API: we measure a hit ratio of 86.66% on 32-bit APIs and 85.32% on 64-bit APIs, reflecting the good heterogeneity of the test suite. The average depth of hooks hit is of 25.73 instructions on 32-bit APIs and 23.29 on 64-bit APIs.

For overheads, we compare the running time of our approach against TOXOTIDAE configured to log arguments like traditional hooking tools do. Building both tools with `-O2` in MSVC and the optimized buffering of SNIPER, we measure an average slowdown of 4.66% (std. dev. $\sigma = 0.0399$) on 64-bit tests and of 7.62% ($\sigma = 0.0765$) on 32-bit tests. Hence, the increased number of hooks results in modest run-time overhead increases on call-intensive tests. This answers **RQ3** and concludes our evaluation.

6 Other Related Works

Binary Analysis. In the vast universe of binary-level program analysis techniques proposed over the years, our work shares loose analogies with techniques that have been explored, among others, for pointer analysis [20], pinpointing memory vulnerabilities in unknown layouts [32], or recovering symbolic information from stripped binaries [3]. However, we are unaware of techniques directly applicable to the tasks of Section 4.

Stolen Code. Countermeasures for stolen code techniques are given in API Chaser [19]. The authors propose techniques based on pre-boot disk tainting and code taint propagation to tag instructions according to their source: *benign* for system code and known applications, *api* for API code, and *malware* if from the untrusted program under analysis. The method intercepts stolen code sequences as the stolen code will have a benign tag and the control transfer to it will have a malicious tag. Documented ways to hinder this approach include issuing calls via code reuse gadgets, as they would be tagged as from a benign location, and imprecisions of taint analysis with implicit flows. API Chaser comes with important setup costs and overheads due to the taint analysis. Our approach is more efficient in both respects. API Chaser would handle our stolen code extension but be defeated by the TOCTTOU attack.

TOCTTOU Attacks. These attacks are well-known to users of UNIX-like system. For example, Wei et al. [33] exploit a race condition between the code responsible for checking file existence and permissions and the code for opening a file, resulting in gaining access to files restricted to superusers only. Two recent industry presentations [14, 15] leverage a race condition between a system call invocation and the Linux kernel when copying memory content from user space to kernel space. An attacker may use the condition to hide the parameters for a few system calls. Our attack works instead in userland, systematically identifying races and categorizing them for exploitability.

Schwarz et al. [30] propose microarchitectural techniques to detect double-fetch operations and a fuzzer to narrow down the exploitable cases, attacking trusted execution environments and system calls. Our approach is different, as we study code statically and do not need a double-fetch vulnerability to attack a location, but it may be interesting to look into similar exploitation techniques for generalizations of our attack.

7 Limitations and Future Works

We acknowledge the following limitations for the research presented in this paper.

As we rely on static analysis for analyzing Windows DLLs, we may miss locations or model wrong ones due to imperfection in the CFG reconstruction phase, when modeling the effects of specific instructions in `compute()`, or when PDB files do not contain prototype information for internal calls to helper functions. We discussed an example of modeling imprecision in Section 5; more may occur when considering other libraries or, more generally, code generated for other platforms or systems.

For stolen code attacks, we left out cases of complete code copies, considering them inconvenient for an attacker. In particular, recursive stealing of internal calls is necessary for stealth (or those would show as from program code and hooking tools would log them) and comes with issues that we discussed in Section 3.1. Nonetheless, complementary measures are a possibility, such as using the code tainting of API Chaser [19] or shepherding memory and file content transfers involving Windows DLLs.

For TOCTTOU attacks, one aspect that we did not cover in the presentation is when an API monitoring solution may re-inspect the initial storage of parameters upon API return. This may expose attacks to locations of type ❶ at API return time. Therefore, the attacker would need to restore the initial fake values at such locations. While for API entry events we came up with sentinel values or delay calibration loops, the only strategy that we foresee for an attacker is to halt execution with a breakpoint on an instruction within the API epilogue or one nearby that dominates it.

More evaluation and calibration is needed for threat actors to deploy our TOCTTOU attack at a scale, also depending on their reluctancy to occasionally fail and reveal the intended API arguments. This observation mainly applies to type-❶ locations when the attacker prefers not to use a deterministic execution interception mechanism and no sentinel value is available, leaving the delay calibration loop as only viable option. Nonetheless, the goal of this paper is showing that these attacks are possible and that, as we showed, can be countered in a principled way.

Finally, we are hopeful that the techniques presented in this paper may be helpful for other tasks. We hinted at a potential application for debug information recovery in Section 4.6. Another interesting opportunity could be to study how our deep hooks

may be used to raise the bar to unhooking attacks to anti-virus and EDR products and to similar approaches that disable key Windows components like AMSI and ETW.

Acknowledgements. We thank our anonymous reviewers for their comments. We also thank Federico Palmaro and Franco Gioia for their feedback on our design. This work was supported by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through project SERICS (PE00000014).

References

1. Apostolopoulos, T., Katos, V., Choo, K.K.R., Patsakis, C.: Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. Future Generation Computer Systems **116**, 393–405 (2021)
2. Assaiante, C., D'Elia, D.C., Di Luna, G.A., Querzoni, L.: Where did my variable go? poking holes in incomplete debug information. p. 935–947. ASPLOS 2023, ACM (2023)
3. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) Compiler Construction. pp. 5–23. Springer (2004)
4. Bhattacharyya, A., Tesic, U., Payer, M.: Midas: Systematic kernel TOCTTOU protection. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 107–124. USENIX Association (Aug 2022)
5. Capstone Engine: Capstone, The Ultimate Disassembler, https://www.capstone-engine.org/
6. Chaitin, G.J.: Register allocation & spilling via graph coloring. In: Proc. of the 1982 SIGPLAN Symposium on Compiler Construction. p. 98–105. SIGPLAN '82, ACM (1982)
7. Chen, R.: Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?, https://devblogs.microsoft.com/oldnewthing/20221109-00/?p=107373
8. Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X., Marion, J.Y.: Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. p. 395–411. CCS '18, ACM (2018)
9. D'Elia, D.C., Coppa, E., Nicchi, S., Palmaro, F., Cavallaro, L.: Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). p. 15–27. Asia CCS '19, ACM (2019)
10. Di Luna, G.A., Italiano, D., Massarelli, L., Österlund, S., Giuffrida, C., Querzoni, L.: Who's debugging the debuggers? exposing debug information bugs in optimized binaries. p. 1034–1045. ASPLOS '21, ACM (2021)
11. D'Elia, D.C., Coppa, E., Palmaro, F., Cavallaro, L.: On the dissection of evasive malware. IEEE Transactions on Information Forensics and Security **15**, 2750–2765 (2020)
12. D'Elia, D.C., Nicchi, S., Mariani, M., Marini, M., Palmaro, F.: Designing robust API monitoring solutions. IEEE Transactions on Dependable and Secure Computing **20**(1), 392–406 (2023)
13. Guilfanov, I.: IDA Pro, https://hex-rays.com/ida-pro
14. Guo, R., Zeng, J.: Phantom attack: Evading system call monitoring (2021), https://www.youtube.com/watch?v=yaAdM8pWKG8, DEFCON 29
15. Guo, R., Zeng, J.: Trace me if you can: Bypassing linux syscall tracing (2022), https://www.youtube.com/watch?v=yFl_ScKA300, DEFCON 30
16. Hasherezade: Tiny tracer, https://github.com/hasherezade/tiny_tracer
17. Iwamoto, K., Wasaki, K.: Malware classification based on extracted API sequences using static analysis. In: Proc. of the 8th Asian Internet Engineering Conference. p. 31–38. AINTEC '12, ACM (2012)
18. Kalendarov, I.: Blindside: A New Technique for EDR Evasion with Hardware Breakpoints, https://cymulate.com/blog/blindside-a-new-technique-for-edr-evasion-with-hardware-breakpoints

19. Kawakoya, Y., Iwamura, M., Shioji, E., Hariu, T.: API chaser: Anti-analysis resistant malware analyzer. pp. 123–143. RAID '13, Springer (2013)
20. Kim, S.H., Zeng, D., Sun, C., Tan, G.: BinPointer: Towards precise, sound, and scalable binary-level pointer analysis. p. 169–180. CC 2022, ACM (2022)
21. Lee, J.h., Han, J., Lee, M.w., Choi, J.m., Baek, H., Lee, S.J.: A study on API wrapping in themida and unpacking technique. Journal of the Korea Institute of Information Security & Cryptology **27**(1), 67–77 (02 2017)
22. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. p. 386–395. ACSAC '14, ACM (2014)
23. Linn, C., Debray, S., Andrews, G.: Stack analysis of x86 executables. Tech. rep., Department of Computer Science, University of Arizona (1212)
24. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. p. 190–200. PLDI '05, ACM (2005)
25. Microsoft: API stack usage, https://learn.microsoft.com/en-us/cpp/build/stack-usage?view=msvc-170
26. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc. (1998)
27. Nektra: Spystudio, https://www.nektra.com/products/spystudio-api-monitor/
28. Rohitab: API monitor, http://www.rohitab.com/apimonitor
29. Roundy, K.A., Miller, B.P.: Binary-code obfuscations in prevalent packer tools. ACM Comput. Surv. **46**(1) (Jul 2013)
30. Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., Mangard, S.: Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. p. 587–600. ASIACCS '18, ACM (2018)
31. Shudrak, M., Bruening, D., Testa, J.: Drltrace, https://github.com/mxmssh/drltrace
32. Wang, H., Xie, X., Lin, S.W., Lin, Y., Li, Y., Qin, S., Liu, Y., Liu, T.: Locating vulnerabilities in binaries via memory layout recovering. p. 718–728. ESEC/FSE 2019, ACM (2019)
33. Wei, J., Pu, C.: TOCTOU vulnerabilities in UNIX-Style file systems: An anatomical study. In: 4th USENIX Conference on File and Storage Technologies (FAST 05). USENIX Association (Dec 2005)

A Appendix

Figure 6 shows average stolen code depths and their median absolute deviation computed for all APIs in a DLL, excluding cases of trivial redirections, as opposed to the analysis of Figure 2 that features only TOCTTOU-vulnerable APIs.

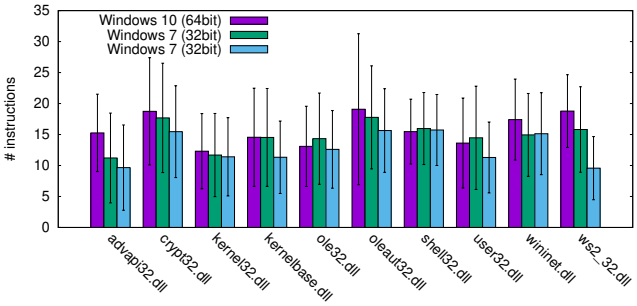


Fig. 6: Stolen code depth with mean absolute deviation bars.