Mitigating Temporal Memory Safety Errors in the Linux Kernel

by

Jake Backer

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

April 2023

APPROVED:

_____

Professor Robert J. Walls, Major Thesis Advisor

_____

Professor Craig Shue, Thesis Reader

Abstract

Temporal memory safety vulnerabilities can allow attackers to escalate privileges on Linux based devices. This paper presents two solutions to temporal safety vulnerabilities. First, we present the *Bounce Allocator*: A pool-based memory allocator designed to mitigate temporal memory safety errors on ARMv8.5-A based Linux devices. Other solutions do not effectively mitigate temporal memory safety errors or have large memory and performance overheads that makes them unsuitable for production environments. The Bounce Allocator achieves entropy comparable with other solutions while using significantly less memory and having improved runtime performance. The Bounce Allocator is implemented on top of an existing allocator to help preserve kmalloc caching performance.

This paper also presents *Tag Exclusion Sets*: A simple solution that deterministically prevents a subset of temporal memory safety attacks in the Linux kernel. This solution has little runtime overhead and no memory overhead and requires little change to the memory allocator.

Acknowledgements

I would like to thank my advisor, Robert Walls, for his support and guidance throughout this work and my studies. I also thank John Criswell and Ziming Zhao for their valuable advice and collaboration this year. I thank Craig Shue for his support and advice throughout my time at WPI. Thanks to my family, friends, and loved ones for their unwavering support. Finally, I thank my parents and grandparents for giving me the opportunity to make it to where I am today.

Contents

List of Figures

Chapter 1

Introduction

The Linux kernel is used on many types of devices and architectures from mobile devices, application servers, and even personal computers for some users. Temporal memory safety errors that take advantage of memory allocators are widespread. DirtyCred is a novel exploitation method for escalating privileges on Linux-based systems that takes advantage of temporal memory safety errors. DirtyCred swaps unprivileged and privileged kernel credentials to gain further access on a system [7].

Lin et al. proposed a solution to the DirtyCred problem involving separating credential structures into non-overlapping memory regions based on the structure's privilege level [7]. This approach makes it hard for the Linux kernel to cache and reallocate objects. This hurts the performance of the memory allocator. Kernel Address Sanitizer (KASAN) is a feature built into the Linux kernel to dynamically find spatial and temporal memory safety errors. In *generic* mode, KASAN has high memory and performance overhead [6]. KASAN also has a hardware tag-based mode that is significantly faster, but only distinguishes between freed and allocated memory. This causes reallocated memory chunks to be accepted. Alternative memory allocators such as DieHard partially mitigate this problem by randomizing the placement of new allocations in a large segment of memory.

Though DieHard provides significant security benefits, it has very high memory overhead compared to conventional allocators [2].

In this paper, we propose alternative solutions that, depending on the scenario, can be used to provide significantly increased security or performance compared to the solution proposed by Lin et al. First, we propose a solution that allows us to protect kernel memory structures with high probability in all scenarios with little performance impact by adding a level of indirection to pointer dereferences. Since smaller allocations are easier to protect with randomization based approaches, a level of indirection to pointers allows us to achieve higher entropy with the same amount of memory.

When designing this allocator, we aimed to allow setting a configurable amount of minimum entropy to allow system administrators to trade off memory overhead for an increased level of entropy. We also wanted to ensure that allocations and frees are done in constant average time. Finally, we must ensure that attempts to probe the heap fail with a high degree of certainty.

We also describe a solution that takes advantage of ARM Memory Tagging Extensions (MTE) to deterministically prevent privilege escalation on ARM-based Linux systems. We reserve a set of tags for privileged data, a set for unprivileged data, and one tag for freed data. By doing so, an invalid free would invalidate pointers and it would not be possible for a privileged object to be allocated with the same tag as the previous, unprivileged, object. This simple solution takes advantage of fast, hardware level features in ARM devices to deterministically prevent privilege escalation.

Our contributions are as follows:

- We designed a system that provides efficient temporal memory safety error mitigation for the Linux kernel. This system can be used to protect any memory structure that uses the kernel memory allocator to allocate instances and is not restricted to protecting against privilege escalation attacks.

- We built an analogue to the system, designed to evaluate performance, in user space that works on top of the glibc memory allocator instead of the kernel memory allocator.

- We evaluated the performance and memory overhead of analogue design. The analogue of the Bounce Allocator incurred a 50.31% performance overhead when compared to the base glibc memory allocator without modifications. The analogue also incurs a 8 byte overhead per allocated object, plus a static amount of memory used for the ready list.

- We designed an alternative solution that deterministically prevents privilege escalation vulnerabilities, such as those shown in the paper by Lin et al., on ARM based systems with little performance overhead [7].

Chapter 2

Background

In this section, we talk about ARM architecture features used by the Bounce Allocator and Tag Exclusion Sets designs.

2.1 ARM Memory Tagging Extension

Introduced in ARMv8.5-A, the ARM Memory Tagging Extension (MTE) adds hardware features to increase spatial and temporal memory safety [5]. MTE consists of two major components: Address tags and memory tags. All memory is broken into 16 byte granules. Each granule has four bits of metadata associated with it. The actual location of this metadata is determined by the hardware, but it is treated as in-band metadata in software. This in-band metadata stores the *memory tag*. MTE takes advantage of Top Byte Ignore (TBI), a feature of Armv8-A that ignores the top byte of all addresses during translation to allow for in-pointer metadata. With TBI, MTE stores the *address tag* in the top byte of pointers. Whenever a pointer is dereferenced, the address tag and memory tag for the memory the address points to are compared in hardware. If they are equal, the dereference occurs. If they do not match, an exception occurs according to the configuration, though

it is recommended to cause a process crash upon receiving an exception in a production environment [5]. This is based on a lock-and-key approach to providing memory safety.

MTE tags can be controlled by software and can be used to store any metadata, but are designed to be used by memory allocators to distinguish between allocated and freed memory efficiently.

Chapter 3

Bounce Allocator

This chapter discusses the potential threats that our system aims to mitigate, our design goals, as well as a design to solve temporal memory safety issues in the Linux kernel, as presented in the DirtyCred paper [7].

3.1 Design

We designed the Bounce Allocator to protect Linux kernel memory structures from heap based temporal memory safety attacks. Below, we describe the threat model, the design goals, and the design of the Bounce Allocator's primary components.

3.1.1 Threat Model

In our threat model, we are focusing on ARM-based Linux systems. Nearly all smartphones use ARM-based processors. More recently, Apple Silicon based Mac devices also use ARM-based processors. We assume that the adversary has access to an an unprivileged user account and a heap memory error in the Linux kernel. We are focusing on ARM based processors with MTE support. We discuss in Section 3.6 how this may be

done without MTE.

We assume the attacker can trigger object allocations in the kernel, deference allocated objects, and free allocated objects. Further, we assume that the attacker can invalidly free objects with a heap vulnerability to create a dangling pointer. We assume that the attacker will create a dangling pointer to point to another valid memory structure. They cannot craft arbitrary pointers and cannot arbitrarily write to memory. We consider the attacker's goal to perform a lateral movement attack and gain access to a different, non-root user.

Temporal memory errors are important to address because any heap memory error can be pivoted to one useful to replace a credential structure in memory and therefore escalate privileges [7]. This type of attack is demonstrated in DirtyCred. DirtyCred is an exploitation method that takes advantage of existing heap vulnerabilities to swap valid credential structures after they have been checked [7].

The details of this technique vary depending on the exact vulnerability that is being exploited. As a high level example, we can analyze the general flow of exploiting CVE-2021-4154 [8]. This is shown in Figure 3.1. In essence, this vulnerability allows the attacker to replace a valid, unprivileged structure in memory with a valid, privileged structure. In this case, the attacker opens an unprivileged file, triggers the vulnerability, and replaces it with a privileged file such as /etc/passwd. This allows the attacker to read or write from the privileged file while acting as an unprivileged user.

Many Linux systems have privileged users other than root. The solutions presented in Chapter 4 and the DirtyCred paper are effective at preventing privilege escalation attacks, where an unprivileged user gains access to the root account on the system, but these solutions do not prevent a lateral movement attack.

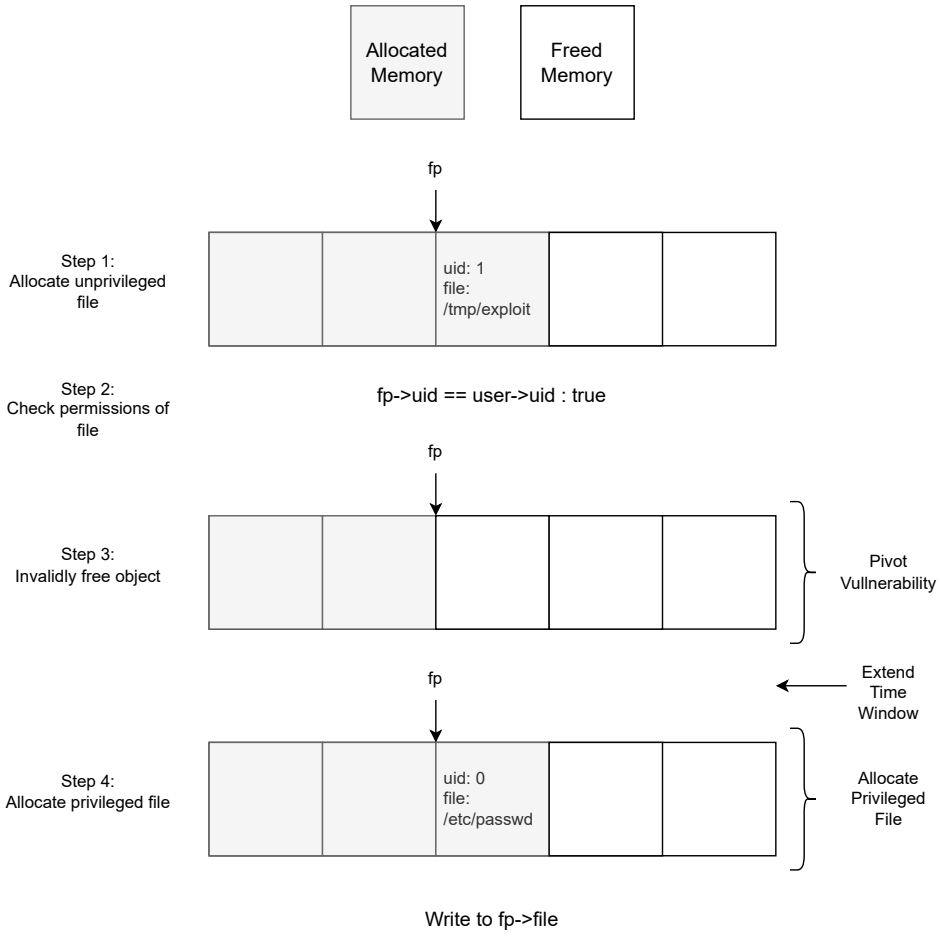

Figure 3.1: Overview of the DirtyCred attack and CVE-2021-4154. This demonstrates the three phases of the DirtyCred attack as well as the details for a specific vulnerability.

3.1.2 Design Goals

We designed an allocator to meet the following goals, in no particular order. By meeting these goals, we mitigate the threats defined in the Threat Model. We discuss how the Bounce Allocator meets these goals below.

Goal 1: *The attacker cannot directly control which slot will be selected*

If the attacker can control the slots provided by the allocator, they can cause the allocator to provide a slot that had been recently invalidly freed by the attacker.

Goal 2: *The allocator provides quantifiable minimum guarantees in preventing an attacker from probing a slot*

If the attacker has the ability to reliably probe a slot, they can perform allocations until they allocate to the spot pointed to by a dangling pointer.

Goal 3: *The allocator provides quantifiable minimum guarantees regarding the level of entropy when selecting a slot*

Goal 4: *Minimize the amount of space used to manage the heap*

Goal 5: *Ensure consistent performance regardless of the number of objects allocated*

If performance slows down when large numbers of objects are allocated, the Bounce Allocator may not be viable in some use cases.

Goal 6: *The allocator provides low runtime overhead for instrumentation*

The Bounce Allocator should use very simple instrumentation to ensure dereferences do not cause a significant performance impact.

Goal 7: *Maintain caching performance and properties of existing allocators*

The Bounce Allocator should not significantly impact the underlying allocators in such a way as to cause a decrease in caching performance.

3.1.3 Design Overview

We built a pool-based allocator that works on top of existing memory allocators, satisfying Goal 7. Our system consists of a large region of memory called the *bounce table*[1] and a fixed sized array known as the *ready free list*.

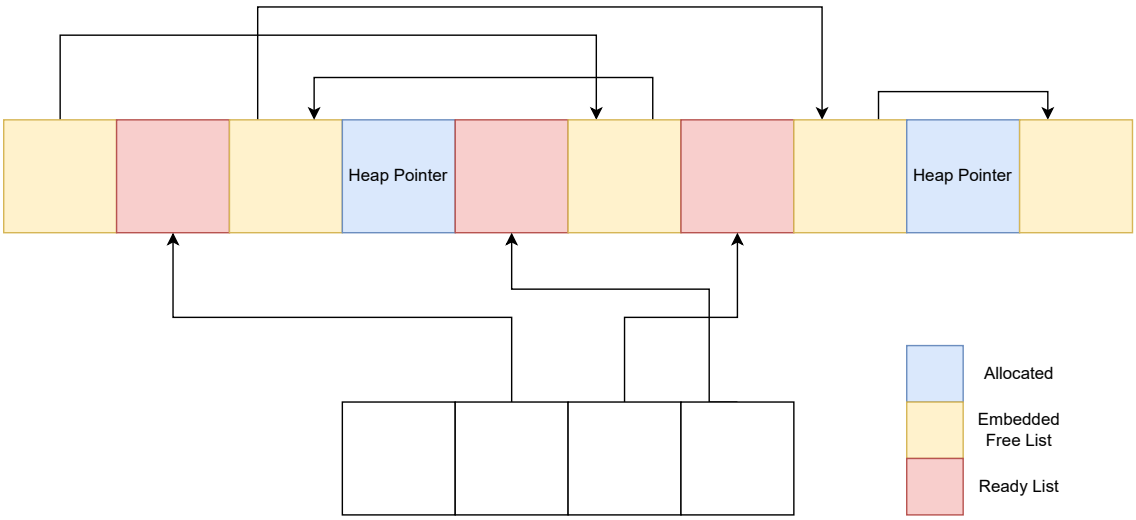

Figure 3.2: The top structure represents the bounce table and the bottom structure represents the Ready Free List. Each arrow points from the table entry containing the pointer to the address it points to. This diagram shows a potential state of the allocator after one table entry from the ready list has been allocated.

On allocation, the allocator chooses a table entry from the ready free list, requests an allocation from the underlying allocator, and stores the allocated pointer in the chosen table entry. When the ready list is half empty, the list is refilled with table entries from the bounce table and the contents are shuffled. This probabilistically prevents reuse of table entries, making it more difficult to perform attacks such as DirtyCred. Since the table entries we are randomizing are small, we can achieve significantly higher entropy with lower memory overhead compared to other randomization based allocators, such as DieHard or the Scudo Allocator [3] [9].

Upon freeing an object, the table entry is tagged as free and is added to the embedded

[1] Also known as the *bounce house*

free list within the bounce table. This prevents the use of dangling pointers due to the tag mismatch. Since table entries are allocated probabilistically, there is an unknown amount of allocations until reuse of the table entry. Additionally, the embedded free list acts as a quarantine list and delays the use of objects as much as possible without wasting memory. This makes it significantly more difficult to perform attacks such as DirtyCred due to the inability to probe table entries and the unknown nature of the randomized table entries.

When objects are dereferenced, the memory and address tags are checked in hardware to ensure the pointer is valid. This allows the allocator to verify the table entry is allocated with minimal overhead. Dereferencing the object a second time will result in the actual structure. With the Bounce Allocator, ARM memory tags are used which cause dangling pointers to become invalid and cause a fault upon derefernece, making it impossible to probe the pointer.

3.1.4 Bounce Table

The bounce table is broken into 16 byte table entries. The Bounce Allocator must distinguish between previously used and new table entries. This could be achieved by tagging data with ARM MTE tags, or by using a separate bit map. Each table entry is always in one of three states: New, allocated, or freed. In our current implementation, these states are encoded in MTE tags which allows the hardware to ensure that pointers never access the wrong state. This incurs minimal runtime overhead and no memory overhead. New table entries have never been allocated. Freed table entries were allocated previously and have since been freed. Allocated table entries contain heap pointers provided by the underlying memory allocator.

When allocating an object in the Bounce Allocator, we request an allocation from the underlying allocator and store its address in the table entry chosen. We then return a pointer to the table entry. This layer of pointer indirection gives the Bounce Allocator

greater control of the pointers associated with allocations without modifying the underlying allocator. Upon freeing a table entry, it is marked with the free tag. This causes any dangling pointers to the table entry to generate an exception when dereferenced. This prevents Use After Free vulnerabilities from being used and satisfies Goal 2.

The Bounce Allocator reserves a large segment of address space at initialization for the bounce table, but only allocates pages as needed. By doing so, the system will not physically assign memory to the requested regions that are not allocated. The purpose of allocating a large, contiguous segment of address space is to reduce calculations needed to find fragmented chunks of the heap.

3.1.5 Ready Free List

The ready free list contains pointers to the table entries of the bounce table. When filled, the ready list is shuffled to provide the main source of entropy for our allocator. The size of the ready list provides a configurable, minimum amount of entropy provided by the system, satisfying Goal 3. The randomized order of the ready list causes the order in which table entries are allocated to be probabilistic. This makes the reuse of table entries unpredictable with a minimum level of entropy, effectively preventing dangling pointers from pointing to newly allocated, attacker controlled memory, satisfying Goal 1. The ready free list also allows allocations to be completed in constant time on average to partially satisfy Goal 5.

The Bounce Allocator makes use of amortized data structures and algorithms. The ready free list allows the Bounce Allocator to delay costly operations of randomizing order and copying data to achieve constant time average performance. The Bounce Allocator uses the Fisher-Yates algorithm for shuffling data, allowing for $O(N)$ time complexity.

3.1.6 Embedded Free List

The bounce table also contains an embedded free list of objects, where a freed table entry contains a pointer to the next freed table entry. This allows us to conserve memory while tracking all previously used objects. This list also preserves the order in which objects are freed, which allows for delaying the reuse of objects until it is necessary to reuse them for memory saving reasons. When an element is freed, the Bounce Allocator adds it to the tail of the embedded linked list. This is done by writing the address of the table entry to the previously freed table entry. This also helps prevent memory sparsity as described in Section 3.1.7. Since this is integrated into the free space in the bounce table, this minimizes the space used to store this metadata, therefore satisfying Goal 4. Adding to the embedded free list can also be completed in constant time to solve Goal 5.

3.1.7 Limitations

There are several limitations in the prototype of the Bounce Allocator. First, the extra layer of memory accesses increases pressure on the Translation Lookup Buffer (TLB) and CPU caches. This could significantly increase the performance impact in real world applications, but initial evaluation shows promising results as seen in Section 3.2. Our current design makes use of huge pages to reduce the strain on the TLB, but future work should be completed to investigate the performance impact [4].

Due to the randomized nature of the Bounce Allocator, large spikes in the number of allocations could result in sparse pages, where very few objects are allocated in a given page at any given time. This increases the overall memory overhead. A potential solution to this is marking pages that have few objects allocated as *ready to be freed* and deferring the allocation of objects in that page by tagging the table entry and moving the pointer to the end of the linked list. This increases the chance all objects in a given page will

be freed, allowing for the whole page to be released, but still falls short when there are persistent objects that are not freed. This is not a problem unique to the Bounce Allocator. Other allocators, such as the Linux SLAB allocator, also have this issue [1]. At a basic level, the SLAB allocator handles this by freeing pages that are empty, but there is also support for callbacks to free pages more aggressively if the OS requests more memory.

3.2 Performance Evaluation

We estimate that the primary source of overhead will be the additional pointer dereference on every object dereference. To serve as a rough estimation of this overhead, we evaluated an analogue of the bounce table to test this additional instrumentation.

Another consideration to make is the size of the ready list for the system. Depending on the frequency that objects are allocated, the size of the ready list can affect performance. If objects are frequently allocated but the ready list is small, then the list must be refilled more frequently. This may cause more slowdowns, however we leave evaluations to future work.

3.2.1 Methodology

We modified a simple merge sort program to use an analogue of the bounce table. We chose this as our benchmark application since it involves a significant amount of allocations and pointer deferences. Since this application contains very few other operations, we are considering this a worst case application for our benchmarks. This program allocates 8000000 objects with random integer values and performs merge sort on them, measuring the time it takes to complete these operations in milliseconds using the gettimeofday function.

To represent the code with the Bounce Allocator, we created an analogue of the

bounce table that contains a large array of table entries. On allocation, malloc allocates the space to store the actual object and the address of allocation is stored in the chosen table entry. Then, the table entry's address is returned. We manually added the instrumentation to follow the bounce table entry. We ran the unmodified baseline and the version with the Bounce Allocator 1000 times and averaged the results.

3.2.2 Results

In this program, the majority of runtime is spent on memory dereferences of heap objects. Therefore, in the worst case we would expect a doubling of runtime due to the duplicated dereferences, however our results show a smaller overhead. On average, the unmodified baseline took 3643.13 seconds and the transformed design took 5476.01 seconds. This results in an approximately 1.5x performance overhead.

3.3 Memory Evaluation

We estimate the memory overhead of the Bounce Allocator as follows. There are two components to consider for memory overhead: the bounce table and the ready free list. The ready free list is statically sized. The size of the list can be adjusted to provide more entropy and can be decided when implementing this allocator. The size of the bounce table is directly proportional to the maximum number of objects allocated. The bounce allocator adds 16 bytes of overhead per allocated object. This results in an overhead roughly equal to $16N + R$ bytes, where N is the number of objects allocated and R is the size of the ready list. This results in significantly less overhead when compared to other solutions such as DieHard, especially in systems where many larger objects are allocated. Due to its over-provisioning, DieHard requires at least a 2x memory overhead, defined by its over-provisioning factor M.

3.4 Security Evaluation

To evaluate the Bounce Allocator's provided security, we investigate the capable actions of an attacker and show how our system protects against these threats. We show the security guarantees of the system and how they apply to the DirtyCred attack. Here, we consider the attacker's goal to cause a dangling pointer in the Linux kernel to point to a data structure that the attacker chooses.

As described in Section 3.1.1, an attacker has four primitive operations from which temporal memory safety attacks in the kernel are constructed: Allocate, validly free, invalidly free, and dereference. The attacker is limited to these operations due to the barrier between kernel and user space. These operations involve calling syscalls that dereferences, frees, or allocates memory internally. The Bounce Allocator instruments each of these operations to add security guarantees.

Upon dereferencing a dangling pointer, a memory tag check will fail since the corresponding table entry will be marked with the *free* tag while the pointer will still be marked with the *allocated* tag. This failure causes a hardware fault, typically causing a crash in most systems. An attacker cannot probe to check if a table entry has been reallocated without a high chance of failure. This effectively prevents attacks such as DirtyCred [7].

When an object is freed, the table entry is marked with the *free* tag and it is added to the end of the embedded free list. Adding to the end (as opposed to adding to the start) of the embedded free list increases the time until reuse without causing increases in memory overhead. This table entry is only reused after all previous table entries in the embedded free list are used. If an object is invalidly freed, any dangling pointers will have a mismatched tag and do not have the ability to match until the object is re-allocated. This happens in an indeterminate amount of time due to the randomization of the ready free list and the difficult to predict nature of allocating many objects in the kernel.

Finally, requesting a new object from the allocator pulls a table entry from the ready free list. This list is randomized and provides a minimum level of entropy. The entropy provided by the ready free list is as follows:

$$log_2(S); S = \text{Number of free table entries}$$

Due to this, the adversary's likelihood of success without causing a noticeable signal, such as a process crash due to a tag mismatch exception, is $1/S$. The ready free list is refilled and randomized once it is half empty. This ensures there is always a minimum level of entropy, but there will likely be more entropy due to the delayed use of table entries:

$$log_2(R/S); R = \text{Size of the ready list}$$

Overall, the Bounce Allocator makes it significantly harder for attackers to predict and manipulate the allocator to perform temporal memory safety attacks. The system randomizes the ordering of table entries with a configurable, minimum amount of entropy and prevents probing table entries to test if an allocation worked.

3.5 Related Work

3.5.1 DieHard

DieHard is a randomization based memory allocator that provides significant security guarantees at the cost of significant memory overhead. DieHard creates *mini heaps*, which are increasingly large chunks of memory that contain memory objects of exactly one size. Each new mini-heap is a factor M larger than the previous mini-heap. When a mini-heap's ratio of allocated objects to total objects reaches $1/M$, a new mini heap is created. This $O(logN)$ bits of entropy from the allocator [3]. A key feature to DieHard is the fact

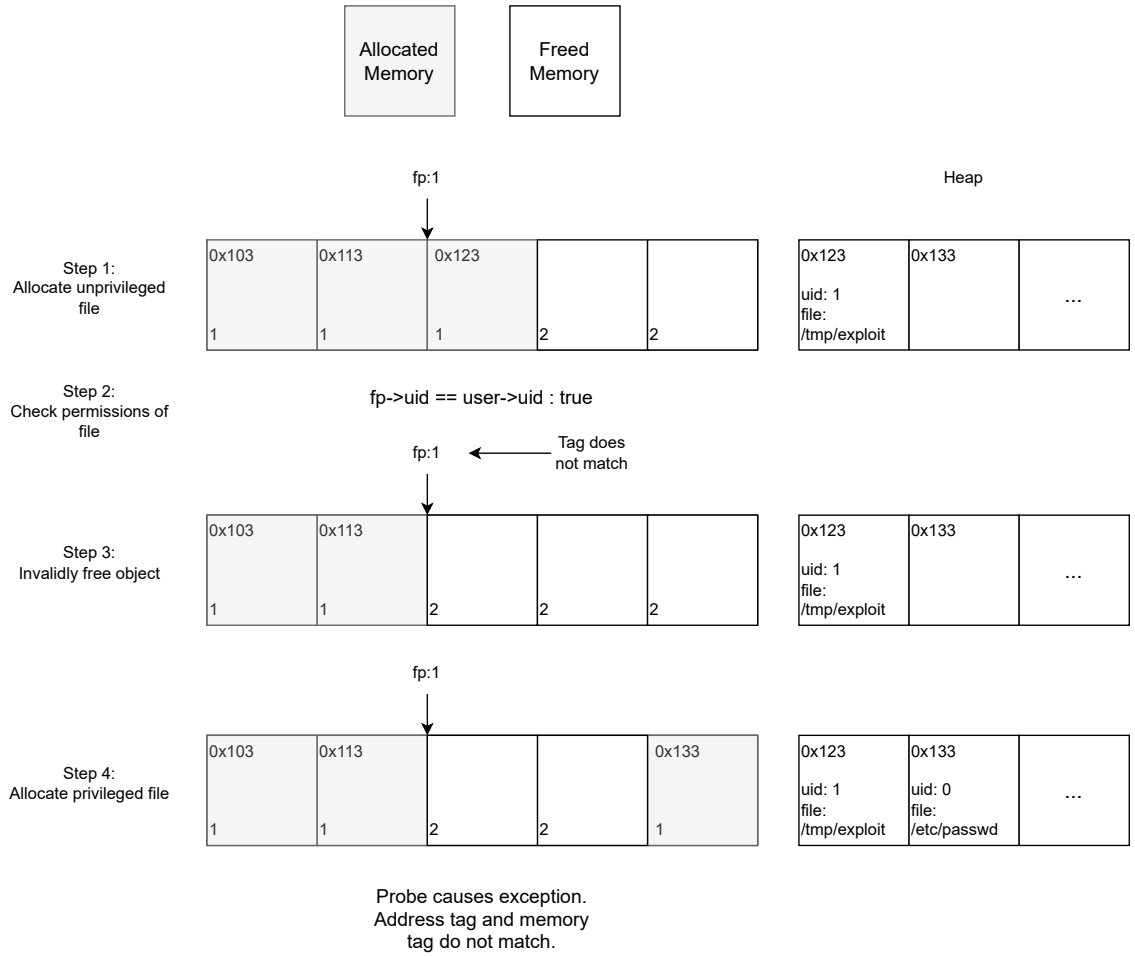

Figure 3.3: The DirtyCred attack and CVE-2021-4154 with the Bounce Allocator. The number after fp: is the address tag and the numbers at the bottom left of each block are memory tags. When the object is freed, the tag is changed. This causes an exception to be thrown upon future probes. The privileged object is randomly allocated into a different table entry, causing the probe to fail.

that the number of free chunks is always proportional to the number of allocated objects, N [3]. This ensures that the probability of returning the most recently freed chunk is at most $1/MN$.

In allocators such as the base DieHard, Scudo, or the Linux SLAB allocator, it is possible to probe if a temporal safety attack occured by dereferencing a dangling pointer. With these systems, it is possible to dereference a pointer and observe its value without crashing the process or otherwise causing a fault behavior. This causes these allocators to fail to meet Goal 2. Since DieHard and Scudo are replacements for the existing memory allocators, they also fail to meet Goal 7.

3.5.2 Kernel Address Sanitizer

Tools such as Kernel Address Sanitizer (KASAN) and other sanitizers can deterministically detect some temporal memory safety errors at the cost of significant memory and performance overhead. KASAN has three different modes that provide different security guarantees and drawbacks. In Generic mode, KASAN uses shadow memory, redzones, and significant compiler instrumentation to detect memory errors. This mode has a 2x runtime overhead and a memory overhead roughly equal to $(1/8\ RAM) + (1/32\ RAM) + 1.5 * NumSlabs$ due to the use of shadow memory [6]. Due to this, KASAN in generic mode is only intended to be used during testing to fuzz for memory safety errors and fails to meet Goals 4 and 6. To prevent immediate reuse, KASAN puts freed objects into a quarantine queue to delay reuse. KASAN's software based tagging mode uses ARM Top Byte Ignore to reduce the memory overhead and the hardware based tagging mode reduces the memory overhead even further and reduces the performance overhead to $< 10\%$, but these modes only distinguish between freed and allocated memory and cannot effectively be used to prevent attacks on live systems and are intended to enable live reporting of memory safety vulnerabilities on live systems. Due to this, KASAN also

fails to meet Goal 1.

3.6 Conclusions and Future Work

In this chapter, we discussed a solution to help mitigate temporal memory safety attacks in the Linux kernel. We focused on DirtyCred: an exploitation method that uses temporal memory safety errors in the kernel along with a race condition to replace credential structures and escalate privileges. We developed a promising solution that makes attacks such as DirtyCred more difficult and provides a minimum amount of entropy with low memory and performance overhead without modifying the underlying memory allocators. We showed that by adding a layer of indirection to pointer dereferences, we can gain stronger control of pointers. We determined that small objects are easier to protect with randomization based methods.

There are several areas for future work to further develop this system into one usable in production environments. First, a better solution to freeing unneeded pages should be investigated. Currently, our solution can result in pages with a small number of used table entries that are persistent and will not be freed. This increases overall memory overhead after a long runtime of this system. A potential area of investigation could be strategic placement of objects that are expected to stay allocated for large periods of time.

Another area for future research is investigating the design and performance of this system implemented without ARM MTE to allow for support on x86 based systems. MTE is currently used to efficiently enforce freed and allocated states as it adds nearly zero performance or memory overhead, but this prevents the allocator from running on systems other than ARMv8.5 or newer. Many personal computing devices and servers are based on x86 which does not have hardware memory tagging support. Implementing the Bounce Allocator without hardware tagging should be investigated to expand support.

The multi-threading safety of the Bounce Allocator should also be investigated. Currently, race conditions are possible on some boundary conditions where one thread may allocate a table entry from the free list while another thread starts re-filling it. A simple lock based solution here would prevent this problem, but would cause significant performance impact and effectively force all allocations to be single threaded. A counting semaphore could be used to improve upon this, allowing all threads to allocate without blocking, but this would still cause blocking when a thread must re-fill the ready free list.

Finally, the Bounce Allocator should be benchmarked in kernel space. Our evaluation of the allocator was done purely in user space to get a rough estimate of the performance impact of the system, but it should be tested in the kernel while running real world applications. This would provide a much more accurate measurement of the performance impact if this were to be implemented in the Linux kernel.

Chapter 4

Tag Exclusion Sets

This chapter discusses Tag Exclusion Sets: Another solution to protect against the DirtyCred attack. Unlike the Bounce Allocator, this solution is deterministic with almost no runtime or memory overhead, however it only addresses the DirtyCred attack and does not address temporal memory safety issues in general.

4.1 Threat Model

Similar to the threat model described in Section 3.1.1, we are focusing on ARM-based Linux systems. For this chapter, we investigate a reduced scope problem where an attacker aims to gain access to the root account specifically. We do not consider lateral movement, where an attacker moves to another non-root account. Further, we only aim to prevent low privilege file structures from being swapped with a high privilege file structure, as shown in the DirtyCred attack [7].

4.2 Design

To combat the DirtyCred attack, we propose a solution taking advantage of ARM MTE to protect Linux kernel structures on 64-bit ARM systems. Our solution, *Tag Exclusion Sets*, assigns tags 0 through 7 for unprivileged objects, tags 8-14 for privileged objects, and tag 15 for freed objects. The SLAB allocator will assign a tag to the region following these rules: Privileged objects will never share a tag with unprivileged objects, objects will never share a tag with adjacent objects, and objects will never share a tag with freed memory. Since unprivileged and privileged objects can never have the same tags, it is impossible for a pointer previously pointing to an unprivileged object to continue to be valid after an object swap.

When memory for a structure is allocated, the allocator sets the memory and address tags. Whenever the pointer is dereferenced, the hardware checks the tags. If the tags do not match, an exception will be thrown. The pointer will no longer be valid when an object is freed or reallocated with an object of a different privilege, as the tags will not match. This deterministically prevents the kernel from using swapped credentials.

Swapping between two different low privilege objects yields a different result. There are only 8 tags for unprivileged objects and 7 for privileged objects. Due to the low number of tags, there is a high probability that an attacker could manipulate the system to perform the DirtyCred attack to swap two objects of the same privilege level.

If the structure is the same privilege as the previous one, there is either a 1/6 chance for unprivileged structures and a 1/5 chance for privileged structures to be allocated with the same tag. The outcome of a same-privilege swap is less severe than a privilege escalation, therefore this is considered somewhat acceptable, but due to the high probability, an attacker could manipulate a system to perform this attack. On some systems, there may be non-root accounts that have similar privileges to root accounts (such as *admin*

accounts). These accounts would not be sufficiently protected by this solution.

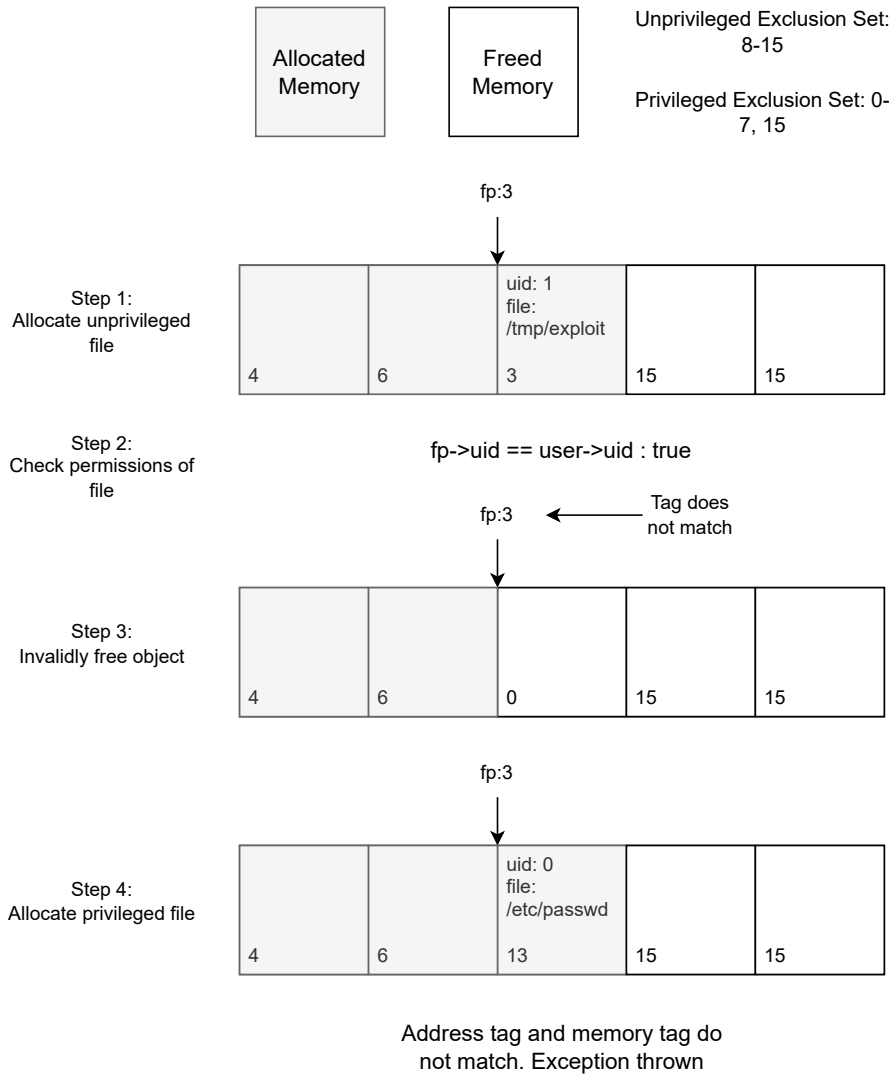

Figure 4.1: DirtyCred attack presented in Figure 3.1 with the Tag Exclusion Set solution applied. The number after ‘fp:‘ is the address tag for the pointer and the numbers at the bottom left of each memory block are the memory tags.

4.2.1 Alternative Solutions

The solution described in the paper by Lin et al. targets the same scope as the solution described here, but it is done entirely in software without hardware tagging support [7]. Their solution separates privileged and unprivileged objects into entirely separate memory regions, preventing them from ever interacting. Though their paper shows there is very little performance overhead with their proposed design, modifications to the kernel allocator can cause issues with object caching in some applications. Their solution also does not provide any protections against lateral movement attacks.

Other solutions that function similarly to our proposed solution do not provide the same security benefits as described here. Kernel Address Sanitizer's (KASAN) hardware tagging mode uses MTE tags to mark freed and allocated memory regions, but it does not distinguish between objects of high and low privilege. Our proposed solution does distinguish between these objects due to its deeper inspection of allocated objects.

4.3 Evaluation

This solution, though not as robust as the solution described in Chapter 3, deterministically prevents privilege escalation attacks following the DirtyCred attack with little overhead. Minimal instrumentation is added to add the tags to the corresponding memory regions and addresses during allocation. No instrumentation is needed to check the tags due to the hardware implementation of MTE. This allows for an efficient implementation of a lock and key mechanism that distinguishes between objects of different privilege level with minimal instrumentation and no memory overhead due to the use of MTE.

Chapter 5

Conclusions

Temporal memory safety attacks, such as DirtyCred, can be used to escalate privileges in the Linux kernel. Current solutions do not effectively mitigate this type of attack or are not efficient enough to be used in production environments. We proposed two solutions that could be used to mitigate two subsets of temporal memory safety issues with little overhead when compared to alternative solutions.

The proposed Bounce Allocator works on top of existing allocators to provide quantifiable, probabilistic security guarantees against temporal memory safety attacks. Though the Bounce Allocator shows promise in providing sufficient levels of performance to be viable for production environments, future work is needed to investigate the performance impact in real world applications. Another limitation of the allocator is the use of ARM MTE. This restricts the use of the allocator to systems using ARMv8.5-A or newer. Future work should be done to expand support to systems without hardware tagging support, such as x86 based systems.

The Tag Exclusion Sets solution provides an efficient way to prevent swapping kernel file structures of different privilege level, as is done in the DirtyCred attack. This solution works with little performance overhead and no memory overhead, but is very limited in

scope compared to other solutions. Future work could be completed to investigate the use of ARM MTE tags to protect kernel structures.

Bibliography

[1] J. Bonwick, "The slab allocator: An Object-Caching kernel," in *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*, Boston, MA: USENIX Association, Jun. 1994. [Online]. Available: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel.

[2] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," en, *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 158–168, Jun. 2006, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1133255.1134000.

[3] G. Novark and E. D. Berger, "Dieharder: Securing the heap," en, in *Proceedings of the 17th ACM conference on Computer and communications security*, Chicago Illinois USA: ACM, Oct. 2010, pp. 573–584, ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866371. [Online]. Available: https://dl.acm.org/doi/10.1145/1866307.1866371.

[4] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," en, in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Williamsburg VA USA: ACM, Mar. 2018, pp. 679–692, ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173203. [Online]. Available: https://dl.acm.org/doi/10.1145/3173162.3173203.

[5] Arm, "Armv8.5-a memory tagging extension," White Paper, Aug. 2019. [Online]. Available: https://developer.arm.com/-/media/Arm%5C%20Developer%5C%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[6] A. Konovalov, *Sanitizing the linux kernel: On kasan and other dynamic bug-finding tools*, Bilbao, Spain, Sep. 2022. [Online]. Available: https://www.youtube.com/watch?v=KmFVPyHyfqQ.

[7] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," en, in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles CA USA: ACM, Nov. 2022, pp. 1963–1976, ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560585. [Online]. Available: https://dl.acm.org/doi/10.1145/3548606.3560585.

[8] [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-4154.

[9] [Online]. Available: https://llvm.org/docs/ScudoHardenedAllocator.html.