FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers

Yves Younan
Talos Security Intelligence and Research Group
Cisco Systems
ndss15@fort-knox.org

***Abstract*—Use-after-free vulnerabilities have become an important class of security problems due to the existence of mitigations that protect against other types of vulnerabilities. The effects of their exploitation can be just as devastating as exploiting a buffer overflow, potentially resulting in full code execution within the vulnerable program. Few protections exist against these types of vulnerabilities and they are particularly hard to discover through manual code inspection. In this paper we present FreeSentry: a mitigation that protects against use-after-free vulnerabilities by inserting dynamic runtime checks that invalidate pointers when the associated memory is released. If such an invalidated pointer is accessed, the program will subsequently crash, preventing an attacker from exploiting the vulnerability. When checking dynamically allocated memory, our approach has a moderate performance overhead on the SPEC CPU benchmarks: running with a geometric mean performance impact of around 25%. It has no overhead when deployed on widely used server side daemons such as OpenSSH or the Apache HTTP daemon. FreeSentry also discovered a previously unknown use-after-free vulnerability in one of the programs in SPEC CPU2000 benchmarks: *perlbmk*. This vulnerability seems to have been missed by other mitigations.**

I. Introduction

Use-after-free vulnerabilities have become particularly widespread and few mitigations exist to protect against them, while even fewer are currently deployed in production environments. The vulnerability class is present in all types of applications and is the result of retaining pointers to memory that has been freed and subsequently accessing these stale pointers. Exploiting other vulnerabilities, such as buffer overflows, has become harder due to mitigations [1]. This has resulted in use-after-free vulnerabilities becoming a significantly more important target for exploitation in recent years.

A study [2] which surveyed vulnerabilities and corresponding exploits for Microsoft products from 2006-2012 found that there has been a significant change in the vulnerabilities that are found and exploited by attackers due to the introduction of mitigations. In 2011 and 2012, the most exploited memory errors for Microsoft products were use-after-free vulnerabilities. It was also the most exploited vulnerability for both Windows Vista and Windows 7 and is the most common type of memory error that occurs in Internet Explorer. Typically, during their monthly patch-cycle, Microsoft will release patches for Internet Explorer, fixing several use-after-free vulnerabilities. Given that Windows Vista and Windows 7 (and Windows 8) are the Microsoft operating systems that have the most mitigations enabled to prevent successful exploitation of whole classes of traditional vulnerabilities, it is no surprise that attackers have shifted their attention to a vulnerability type that currently lacks widespread mitigations.

Use-after-free vulnerabilities are very hard to spot during manual code review as they require knowing the pattern of allocation and deallocation that occurs during a program's execution. The vulnerability is a temporary one, that only exists at particular points in time when the stale pointer presents itself. The stale pointer can be present for a long time in the program, but the actual vulnerability that occurs when the stale pointer is used can occur several functions or hundreds or even thousands of lines removed from the deallocation, further complicating the review process. As such, this type of vulnerability can easily be missed by even the most

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23190

experienced code reviewer. Unlike missing checks for other vulnerabilities, such as buffer overflows or integer errors, there is no easy check that a programmer can add to ensure that the pointer isn't stale. Even explicitly setting a pointer to *NULL* after freeing memory can cause the issue to be missed because a pointer could have been copied to another pointer.

In this paper we present FreeSentry, a mitigation that automatically protects applications against use-after-free vulnerabilities at runtime through compile-time instrumentation. It provides protection by tracking pointers that point to an object (a chunk of dynamically allocated memory or a stack frame) and invalidates these pointers when the object is freed. We achieve this through source code transformation that inserts runtime code to track pointers and invalidates these pointers when the associated object is freed. Our approach is compatible with existing, unprotected compiled code and does not require intervention by a developer. It is also able to protect against use-after-free vulnerabilities that exist in both dynamically allocated memory as well as in automatically allocated memory (i.e., on the stack). While it doesn't require intervention from a developer to be deployed, it does however support developer cooperation to improve performance. For example, it is possible for a developer to manually opt out of protection for a particular function which can result in significant performance increases. It is also possible for developers to manually register a pointer for tracking even in unprotected functions, by simply calling a library function with the address of the pointer that requires tracking. The performance overhead for our approach is moderate: a mean of 25% for the CPU2000/2006 benchmarks, which makes it significantly faster than other approaches that offer full protection against these types of attacks. We also measured performance overhead for frequently used daemons on servers like Apache HTTPD and OpenSSH and show that the overhead for these applications is negligible.

The rest of this paper is structured as follows: Section II describes use-after-free vulnerabilities and how attackers can abuse them. Section III discusses the design of our countermeasure, while Section IV examines our prototype implementation. Section V presents an evaluation of our prototype by benchmarking it and examines the mitigation's effectiveness through a security evaluation. In Section VI we compare our approach to other mitigations, while Section VII discusses combining our approach with a bounds checker. Finally, Section VIII presents our conclusion.

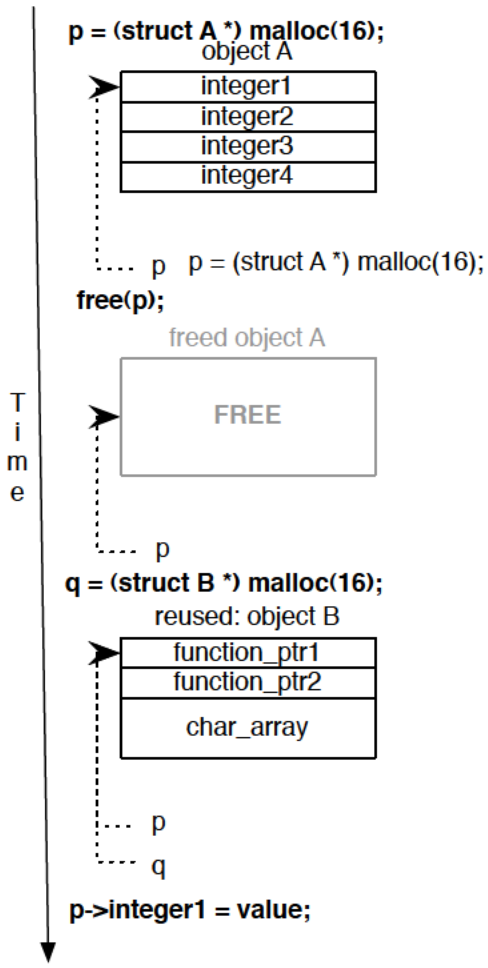

Figure 1. Use-after-free vulnerability

II. Problem Description

A use-after-free vulnerability occurs when a pointer to an object that has been freed is dereferenced. This can result in information leakage but could also allow an attacker to modify unintended memory locations, potentially leading to code execution. When memory is freed, it can be potentially reused by the system when a new memory allocation request is performed. When this happens, one or more objects will inhabit the memory location of the previous object and other datatypes could exist in one object versus another. For example, if we have an object *A* which contains four integers and we have a pointer *p* that points to it. If object *A* is freed but *p* is still used, the program will expect that memory to be inhabited by an object containing four integers. At some

other point the program allocates memory for an object *B* that contains two function pointers and a character array of 8 bytes. Depending on how the memory has been reassigned, the resulting location for object *B* can encompass all or part of the previous memory inhabited by object *A*. However, when pointer *p* is accessed, the program expects to access four integers. If these integers overlap with the function pointers in object B, then attackers could be able to read or write these pointers as if they were integers, potentially allowing them to assign these integers values that would reference their injected code. When the function pointers are executed in object B, the injected code would be called instead of the intended function. Figure 1 provides a graphical representation of this issue.

Listing 1 shows an example program that suffers from two use-after-free vulnerabilities. The vulnerabilities exist on lines 12 and 15 due to releasing of memory at lines 11 and 4 respectively. The issue at line 12 is related to dynamically allocated memory, while the issue at line 15 is due to a pointer to stack-allocated memory which is freed automatically at line 4 when the function returns. The vulnerabilities in this example are simply for illustrative purposes to demonstrate our approach in subsequent sections, they are not exploitable due to the lack of memory reuse between the freeing of memory and the use-after-free vulnerability. If memory was reused as depicted in Figure 1, then an attacker could potentially partially overwrite *function_ptr2* in object *B* with the value 99 (the ASCII value of the character c) at line 12.

Listing 1. A C program that is vulnerable to use-after-free vulnerabilities

```
1   char *retptr() {
2       char p, *q;
3       q = &p;
4       return q;
5   }
6
7   int main() {
8       char *a, *b; int i;
9       a = malloc(16);
10      b = a+5;
11      free(a);
12      b[2] = 'c';
13
14      b = retptr();
15      *b = 'c';
16  }
```

A specialized case of the use-after-free vulnerability is the double free vulnerability where memory is released by calling *free* and that same object is then again freed at some later point in time. This potentially causes the memory allocator to overwrite memory management information stored in the free chunk, which could result in an exploitable state. The only difference between a double free and a regular use-after-free vulnerability is where the dangling pointer is used. In a regular use-after-free, the pointer can be used anywhere, while in the case of the double free, the stale pointer is used together with a call to *free*.

III. Approach

The core idea behind the protection that FreeSentry offers is to link objects back to their pointers. When the memory for the object in question gets freed, then the pointers that still reference the object can be invalidated.

To do this, whenever a pointer is created or modified to point to a new object, the address of the pointer is registered as referring to our object. When the object is freed, the freeing function will look up all the pointers that point to the memory region inhabited by the object. If these pointers still point to our object (they could have been changed by unprotected code), the pointers are invalidated. When a pointer is invalidated, it is made to point to an invalid memory location which will cause the program to crash if it attempts to dereference it. These transformations are done automatically by FreeSentry, no programmer intervention is required.

Listing 2 shows a transformation that is done to the example vulnerable program in Listing 1.

Listing 2. Vulnerable C program protected with FreeSentry

```
1   char *retptr() {
2       labelstack();
3       char p, *q;
4       q = &p;
5       regptr(&q);
6       invalidatestack();
7       return q;
8   }
9
10  int main() {
11      char *a, *b; int i;
12      a = malloc(16);
13      regptr(&a);
14      b = a+5;
15      regptr(&b);
```

```
16          free(a);
17          b[2] = 'c';
18
19          b = retptr();
20          regptr(&b);
21          *b = 'c';
22      }
```

In the transformed program in Listing 2, the following occurs. When a pointer is set to an object, *regptr()* is called with the address of the pointer, registering the pointer as pointing to that specific object. When *malloc()* gets called at line 12, FreeSentry intercepts it at runtime, calls the original *malloc()* function to perform the allocation and registers the memory bounds. When the *free()* function is called, it is also intercepted, the original *free()* function is called and the pointers that have been registered via *regptr()* are invalidated. The *labelstack()* and *invalidatestack()* functions perform the same functions as the intercepted *malloc()* and *free()* functions, but for stack-allocated memory.

FreeSentry does not modify the way that pointers are represented; this means that the program will remain compatible with existing code that may not have been protected with the mitigation. This also allows developers to opt out of protection for specific functions. For example, if a function has been verified manually to be safe, then the function can opt out of registering its pointers. Any memory that the unprotected function frees will still trigger the pointer invalidation as pointers to the freed memory could have been created elsewhere. The same occurs for any memory it allocates: this will still be labelled to allow pointers to the memory to be tracked in other locations. Opting out would only be done to improve performance. Section V-A2 discusses a case study on how opting out of a minimal amount of code can positively affect FreeSentry's performance.

Our main approach is compatible with objects created in all types of memory: whether dynamically allocated, global, static or created on the stack. Of these types of objects, global and static objects do not need to generally be checked for potential use-after-frees because global and static objects are only released when the program terminates. However, having a pointer to stack space that has been reclaimed for other purposes is possible, as the example in Section II demonstrates. This type of vulnerability is rare and can be more easily detected. In fact some compilers will generate warnings when returning a pointer to a stack variable. FreeSentry can run with or without stack protection enabled and we expect that the typical use-case of the mitigation will be to only protect dynamically allocated memory.

A. Supporting data structures

The memory layout for the supporting data structures is presented in Figure 2. The information that is registered about a pointer and where it refers to is called the pointer registration information. To link the pointers to objects, two lookup tables are used. The first one, which we call the object lookup table, is used to look up all the pointer registration information based on the address of an object. The second lookup table is used to look up that same information based on the address of a pointer and is called the pointer lookup table.

The free function uses the first approach to look up the information: when an object is freed, the pointer registration information is found based on the object's memory location using the object lookup table. The code that tracks the pointers uses the second approach: based on the address of a pointer, we can find the registration information.

The first way of looking up information could be eliminated if we transformed the *free()* calls in programs to pass the address of the pointer to the object being freed instead of passing the address of the object by value. However, this approach would mean that any calls to *free()* in unprotected code would no longer be able to invalidate the pointers pointing to that memory. Eliminating the second table is possible, since we can access the value of the pointer being passed into the registration and can thus look up the registration information based on the object. However this would significantly reduce performance as we would need to examine the registration information of all pointers that refer to a specific object to locate the desired pointer's registration information.

We also need to know the start of the object that a pointer is referencing when adding that pointer to the object's lookup table. To do this we use a technique based on the one described in [3]: a unique label is stored for the memory area that an object inhabits when the object is allocated, called a label. When we register a pointer to an object, we look up the label of the object that the pointer references. That label is then used with the object lookup table to find the pointer registration information, to which we then add the new pointer. Objects are a minimum size and can inhabit multiple memory areas

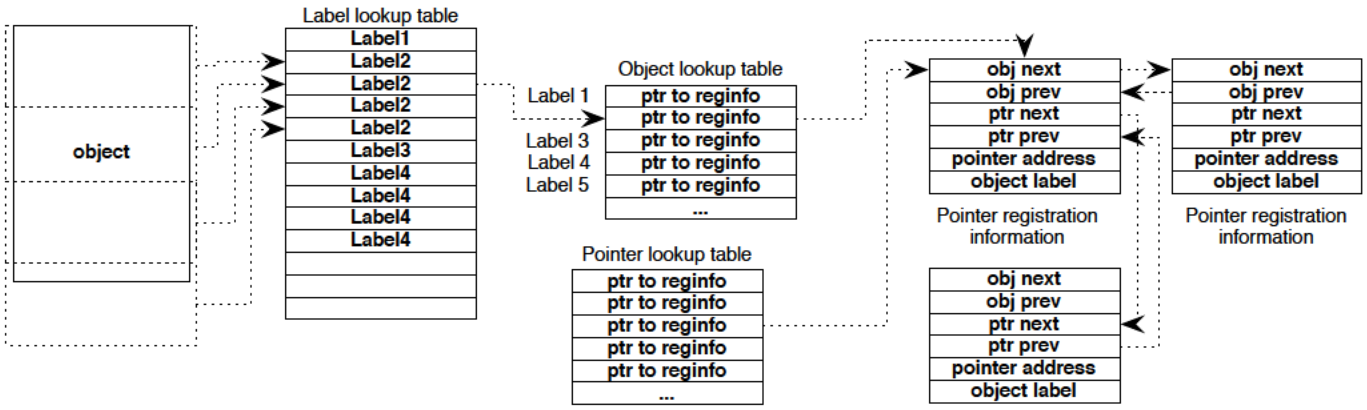

Figure 2. Memory layout of FreeSentry

(called regions) that are multiples of that size. The same label will be used for each of the regions that an object inhabits.

To find an object's label we right-shift the address of the object by the minimum object size which gives us the index of the label in the label lookup table, that label is then used as an index into the object lookup table. To find the pointer registration information based on a pointer to the address, we right-shift the pointer by 4 bits and use that value as an index into the pointer table. Both hash tables are of a fixed size, so a modulo operation is performed to ensure the index points within the hash table. All pointer registration elements contain references to the next and previous elements for both hash tables to deal with collisions. They also contain the label of the object that is being referenced as well as the address of the pointer being registered.

When a pointer is set to a new object, its pointer registration information is looked up via the pointer lookup table, if present, the pointer is set to point to the new object and is removed from the object lookup table by unlinking it from the doubly linked list of object references. If the object already points to the current object then no actions are performed. If the pointer is not present, a new pointer registration information object is created, and added to the front of the respective hash bucket for both the pointer lookup table and the object lookup table. Figure 2 shows the memory layout for FreeSentry.

B. Freeing memory

When memory is freed, we look up the label of the object in the label lookup table. We use this label as an index into the object lookup table, and retrieve the pointer registration information, which contains both the pointer's address and the object label it refers to. If the stored object label matches the label of the object being freed (we might have retrieved a pointer registration for an object that inhabits the location in the hash table due to bucketing), then we check if the pointer is still referring to the object. This is necessary because the pointer may have changed if it was modified in unprotected code. If it is still pointing to our object, then the pointer is invalidated and the pointer registration information is removed.

This approach may introduce dangling pointers of its own: if the memory that a registered pointer lives on is freed, then accessing that pointer via the pointer registration information may cause an invalid pointer access. Because we check the value of the pointer and make sure it still points to our object before invalidating it, this is not an issue in cases where the memory is still accessible. However, this becomes a problem when the page on which the pointer is located becomes invalid. To ensure that the program does not crash as we access pointers, we keep a bit-array that stores the liveness of a page and check if the page is still alive before accessing any pointer stored on it. We keep this bit-array up to date by intercepting calls to *mmap()*, *munmap()* and *mremap()*, and update the bit-array depending on the removal or addition of pages.

C. Reallocating memory

When *realloc()* is called to increase or decrease the size of a memory region, the pointer that is returned could be different from the pointer that is passed in as argument. The only guarantee that *realloc()* gives is that

the old data will remain intact up to the smallest size (i.e., *min(oldsize, newsize)*). Due to this lack of guarantee, any call to *realloc()* should invalidate all pointers to the old object. However, our goal is to remain as unintrusive as possible if there's no potential for harm. As such, we will only invalidate pointers when the new pointer returned by *realloc()* is different from the old pointer passed into the function. If the mitigation is used as a testing tool to detect as many use-after-frees as possible, then it is beneficial to turn on invalidating of all the old pointers.

When *realloc()* allocates new memory, it copies the old data over to the new memory location and subsequently frees the old memory location. Any pointers to the old memory location are now stale. An example of this type of use-after-free can be found in CVE-2009-0749 [4]. FreeSentry found a vulnerability of this type in the *perlbmk* benchmark which is part of the SPEC CPU2000 benchmarks used to test performance in Section V. The details of this vulnerability are discussed in Section V-B1.

D. Stack protection

If stack protection is enabled, we perform labelling of the stack frame when the function is entered and then invalidate any pointers that refer to this stack frame when leaving the function.

There are two complications to this approach. First is the *alloca()* function, which allocates memory dynamically on the stack and as a result causes changes to the stack frame. To support this function, we ensure that all calls to *alloca()* allocate a multiple of our minimum object size and insert a library call after an *alloca()* to update the labelling to include this newly allocated area.

The second issue is due to *longjmp()*, which is a function that will jump to the last place in the code where *setjmp()* was executed, resetting the stack pointer and other registers to the value they held at the time of the *setjmp()* call. These changes in stack pointer need to be tracked by FreeSentry to be able to invalidate the stack frames that have been freed due to the *longjmp()* call. To facilitate this, we intercept calls to this function and invalidate each stack frame separately by walking over the saved frame pointer until we reach the frame we're returning to. This ensures that any pointers that become stale due to the *longjmp()* are properly invalidated.

While FreeSentry supports protecting against dangling pointers to stack-allocated variables, their occurrence are rare. As such, we assume that the typical use case for the mitigation will not include the stack protection: the added performance impact of enabling stack protection is too high relative to the rarity of this type of vulnerability.

E. Pointer arithmetic and out-of-bounds pointers

If simple pointer arithmetic occurs increasing or decreasing the value of a single pointer, then we do not consider this as a change in target object and thus no tracking is added at compile time. This is due to the fact that if no buffer overflows exist, we can assume that an object will stay within the bounds of the object it refers to. Out-of-bounds pointers can be created this way, but they simply will be considered to still point within bounds by our implementation. However these pointers will not be invalidated when the object is freed as we expect the values to be in bounds at the time of deallocation. This provides for maximum compatibility for FreeSentry: if a pointer is changed in unprotected code and now points to a new object, which might be immediately adjacent to the object being freed, then we cannot invalidate it if it no longer points within the bounds of the original object.

If pointer arithmetic occurs where a value is assigned to a pointer based on arithmetic with a different pointer, then an out-of-bounds value can still occur which can cause incompatibility with our approach: if the out-of-bounds pointer points to a new memory location that is subsequently freed, then the pointer will be invalidated. If new pointer arithmetic occurs on this out-of-bounds pointer that would make it go in bounds again, then the result will still be invalid. This type of incompatibility only occurs with programs that generate illegal out-of-bounds values (i.e., not compatible with the C standard) and can be solved by combining our approach with a bounds checker that supports illegal out-of-bounds values. In Section V we discuss a workaround that we used to run the benchmarks on a program that generates these illegal out-of-bounds pointers.

Another possibility that can exist is pointer arithmetic with freed values. For example, one piece of code that FreeSentry would break when used naively is shown in Listing 3:

Listing 3. C program with pointer arithmetic

```
1  int main() {
2    char *a, *b;
3    int difference;
4    a = malloc(100);
```

```
5     b = a + 8;
6     free(a);
7     difference = b - a;
8   }
```

Whether the code in Listing 3 is valid, is arguable: it is valid in C to subtract two pointers that refer to the same object. One might argue that if the object no longer exists, then the pointers can no longer point to the same object and thus one might expect undefined behavior by the compiler. However, even if the memory has been reused, as long as the pointers are not dereferenced, then no exploitable use-after-free vulnerability has occurred. If our mitigation simply invalidated both pointers without taking this possibility into account, some programs might break (in fact, one of the benchmarks, *perlbmk*, performs this exact operation). To achieve maximum compatibility, our mitigation will invalidate pointers by making them point to a reserved area of memory. In our implementation, we assume that the top 1GB of memory has permissions that our user-mode program cannot access. This is the case for both 32-bit and 64-bit versions of both Linux and Windows, where the top areas of memory in a user-mode process are reserved for the kernel. Any access to the kernel address space, will result in a segmentation fault. This allows our implementation to invalidate a pointer by simply setting the first two bits to one (on Windows systems, just setting the first bit to one is enough as the top 2GB of memory is reserved). This allows this type of arithmetic to keep working[1].

F. Pointers copied as a different type

One limitation in our approach is that it does not track pointers that are not copied as pointers, i.e., if a pointer is copied as a different type, this will not be tracked by our approach. This can occur for example, when a programmer calls the *memcpy()* function to copy one area of memory to another. The memory is copied as a void type and not through pointer assignment resulting in the copy not being tracked by our mitigation. While our approach cannot automatically detect the copying, it does allow for a programmer to register the pointers in the new memory area by manually calling the *regptr()* function with the address of the newly copied pointer.

[1]On Linux this introduces a minor compatibility issue, if the pointers used in the arithmetic cross the memory boundary where these bits are flipped: i.e. having a pointer above and below 0x40000000 and a pointer above and below 0x80000000. However this is unlikely to happen in practice: it requires a dynamic memory allocation that crosses this boundary, having one pointer that is set above and one that is set below the boundary and then requires the application to free that memory and subsequently perform a subtraction of those two pointers

G. Unprotected code

Unmodified code that is linked to code that is protected by our mitigation will work without issues. Any pointer assignments and propagation in this code will not be tracked by our approach and any dangling pointers that result from this code will not be detected. However, calls to memory allocation functions will still be intercepted, allowing the correct labelling of newly allocated or reallocated memory and the correct invalidation of tracked pointers. We provide the ability for a programmer to manually opt out of tracking by setting a function attribute. This allows for flexibility when deploying the mitigation, allowing a programmer to improve performance by making sure particular often-called functions are safe. We discuss the impact of selective opting out by a programmer in Section V-A2.

H. C++

Our prototype implementation, discussed in Section IV is aimed at C code due to the CIL framework that we use to transform the code. However, the principles hold the same for C++ as they do for C. Pointers operate in much the same way in C++ as they do in C, so tracking occurs in a similar manner. The smart pointers provided by C++ in the form of *unique_ptrs*, *auto_ptrs*, *shared_ptrs* and *weak_ptrs* are built on top of regular pointers using templates, which means they would all be trackable in the manner discussed in the previous sections. However, given the properties that these smart pointers provide, tracking could be simplified by eliminating some of the checks for conditions that cannot occur on a particular pointer subtype. Dynamic memory handling is similar to C: when *new* or *new[]* is called, the memory is labeled by FreeSentry and when *delete* or *delete[]* is called, the pointers to the memory are invalidated.

IV. Prototype implementation

FreeSentry is implemented using CIL [5]. Whenever a pointer is set to an object, the pointer is registered by calling a library function *regptr()*, which creates a new pointer registration information, which stores the address of the pointer and the label of the object that is associated with it and stores pointers to that information in both in

the pointer and object lookup tables. When the program starts up, we allocate these various memory regions by using a constructor function in the library that is linked by the mitigation.

Our default region size is 32 bytes in our prototype implementation, meaning that all objects must be a multiple of 32 bytes in size. To achieve this, we intercept all calls to the memory allocation functions: *malloc()*, *calloc()* and *realloc()*. We then increase the size of the allocated object to be a multiple of 32 bytes. We then ensure that these functions perform the labelling for the allocated objects to ensure that we can track the object sizes. We also made a minor modification to the memory allocation library to ensure that all objects are aligned on a 32-byte boundary, mostly to simplify our implementation. If stack protection is enabled, then we also label the stack frames at the start of the function and use gcc options to ensure that stack frames start at a 32-byte boundary and are a multiple of 32 bytes in size.

A. Optimizations

There are two major optimizations performed by our approach that both rely on call graph analysis that we perform on the program. We examine which functions a particular function calls and to go through that call chain until we either hit a leaf function or a library call. If along the call chain, any function calls the *free()* function (or any variant function that has the potential to free memory, such as *realloc()*, library functions that free pointers passed into it, etc), we consider that function a leaf function that calls *free()*. We propagate the free-calls through the functions in the program, allowing us to know if any function calls *free()* at any time. To be able to support this modelling, we provide models for the functions in typical system libraries such as *libc*, *libm* and *openssl*: noting whether they potentially free any memory they did not themselves allocate. When we encounter a function that doesn't exist in the program, nor in our model (i.e., a library call that we do not recognize), we err on the side of safety and assume that this function calls *free()* and propagate it accordingly. Note that the model (which simply indicates if a library function calls free or not) is not required for FreeSentry to work, if no model is available then we assume that all unknown functions call free and it simply reduces the potential for optimization. This also means that a developer is not required to ever update the default models as they already provide all the basic information for the default system libraries. If new calls are added without an updated model, then they are simply assumed to call free and functions that call them are not optimized.

The first optimization that relies on this approach simply removes pointer tracking for a local variable if the function does not at any point call free and does not take the address of that local variable. We still track changes to pointers in global and dynamic memory, including copies of local variables to this memory, as well as dereferences of local variables that result in changes to pointers. This provides for significant optimizations because CIL introduces many temporary variables when transforming programs, which simply hold intermediate values for complex calculations. This also allows a further optimization for the stack-based approach: if no addresses of local variables are taken, then a function cannot return a pointer to it's local stack frame, allowing us to remove the labelling and invalidating calls for that function.

A second optimization is to introduce loop optimization. If no function calls that free memory are performed in a loop and there are no unexpected exits out of the loop (i.e., no return statements), then the registration for simple pointer assignments (i.e., where there is no arithmetic or dereferencing on the left hand side value) is moved outside of the loop. Statements that use this pointer value will still be tracked, but since the pointer is overwritten every loop iteration, it is only tracked when the loop ends since it can't become stale in the loop.

Both these approaches are safe, because there can be no releasing of memory due to a lack of function calls that call *free()* in the respective optimized scopes.

A few other optimizations include the fact pointers that are set to point to global memory are not tracked because this memory can never be freed[2]. We also do not update pointer information when simple pointer arithmetic occurs that simply changes the value of the base pointer (e.g., *p++;*) , given that we assume that memory stays within bounds as discussed in Section III-E.

V. Evaluation

In this section we evaluate how FreeSentry performs in terms of performance overhead. We also provide a security evaluation and discussion, showing that the mitigation is able to prevent exploitation of real-world

[2]A rare vulnerability can occur when a module is unloaded, as evidenced by CVE-2010-0425[6]

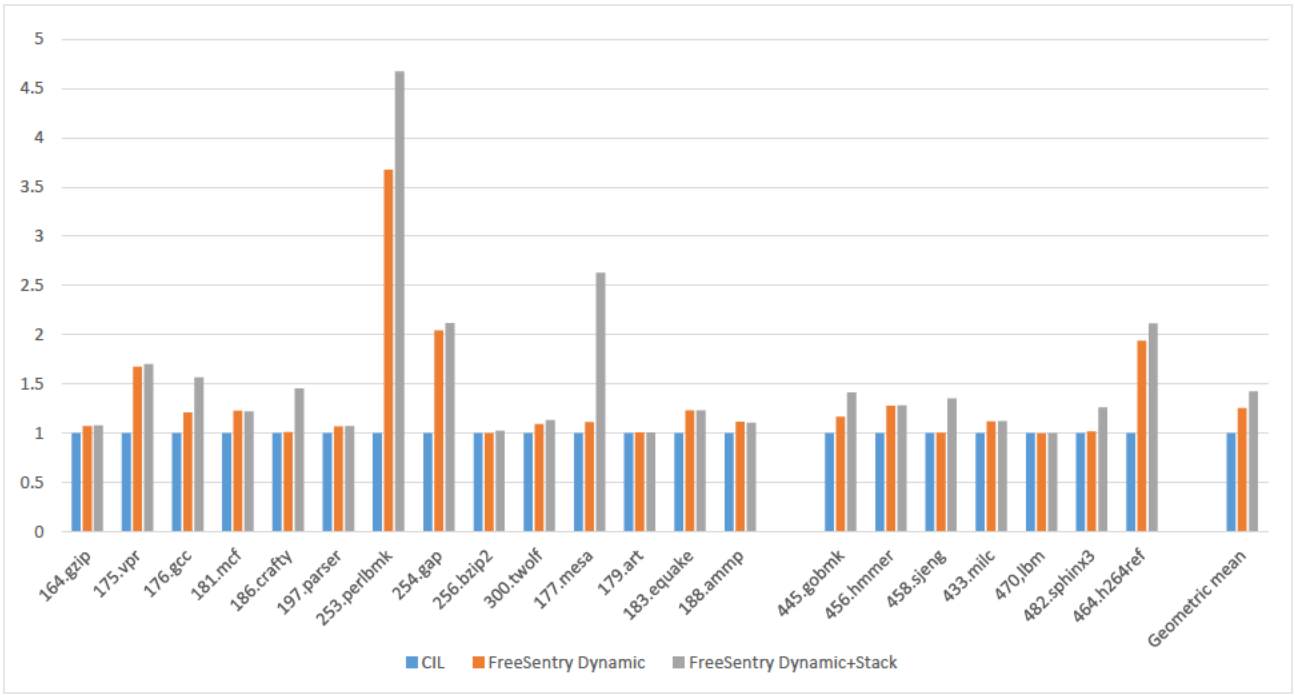

Figure 3. SPEC CPU2000/CPU2006 performance benchmarks

vulnerabilities. All the benchmarks and evaluations in this section were done on a single machine: an Intel Core i7-286QM with 16GB of RAM running the 32-bit version Ubuntu 12.04.03 with kernel 3.8.0-29-generic. All benchmarks were compiled with CIL-1.6.0, which in turn used gcc 4.6.3 with the *-O2* optimization level. Each program was linked with dlmalloc-2.7.2: the original one for our baseline benchmark and the slightly modified one for FreeSentry.

A. *Performance overhead*

In this section we evaluate and discuss the performance overhead of our approach using a number of benchmarks: the SPEC CPU2000 and SPEC CPU2006 benchmarks and a benchmark consisting of a number of popular server daemons.

Figure 3 provides results for the C programs in the SPEC CPU2000 and CPU2006 benchmarks (except for the programs that are already part of CPU2000). The base run is performed by compiling the programs with CIL and subsequently running the resulting programs. We provide the base run using CIL so as to measure the impact of FreeSentry free of interference from tools. The CIL framework introduces very little to no overhead on the SPEC benchmarks when compared to GCC [3], so the use of CIL will have negligible performance impact on real world deployment compared to GCC.

We then measure the runtimes of the same programs protected by FreeSentry. The mitigation results are presented twice, once with the base mitigation which provides protection for the most common case; use-after-frees where a stale pointer refers to dynamically allocated memory. The second results are for FreeSentry with both protection for use-after-frees in dynamically allocated memory as well as on the stack. All tests were run with the reference load supplied by the SPEC benchmarks. Most programs were run unmodified (besides for the automatic checks inserted by the mitigation) for the results provided here, except for *perlbmk* where a minimally modified version that fixes the vulnerability described in Section V-B1 was used for both the base run and the protected runs. One of the benchmarks, *vpr*, creates illegal out-of-bounds pointers [7], [3] that point before the object. To run this benchmark, we modified our mitigation to allocate an extra 8 bytes to every allocated chunk and to increase the returned pointer by 8 bytes. The pointer is decreased again when the memory is freed. Having support from a bounds checker with support for out-of-bounds objects, as discussed in section VII, would allow the program to work without workarounds.

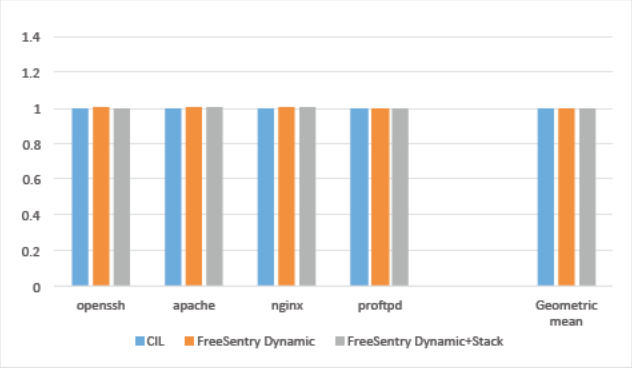

Figure 4. Performance benchmarks for daemons

The geometric mean performance impact for our mitigation for the CPU2000 benchmarks is 29% for the dynamic memory protection and 47% when the stack protection is included. For the CPU2006 benchmarks the overhead is 18% and 32% respectively. With an overall mean overhead of 25% and 42% over both benchmarks. This is a moderate overhead for this type of protection: CETS [8], which also does source-based transformation, has significantly higher overhead. If we compare the 14 programs that are reported in both studies, there is a mean overhead of around 48% for the measurements by the CETS mitigation, while we have a mean overhead of 29% for our mitigation with stack protection turned on (CETS also provides protection for the stack). The overhead for our dynamic protection is 12% for these programs.

The program with the highest overhead in our measurements is Perl and this is mainly due to the significant amount of allocations and deallocations that Perl performs in the benchmarks. However, the vast majority of programs protected by our mitigation have a much lower impact on performance, in many cases the impact of dynamic protection is negligible.

As discussed in Section III-D, we believe that the mitigation is best used without stack protection as these types of vulnerabilities are rare and the benefit of protection for these types of vulnerabilities does not outweigh the added performance impact.

1) Daemons: To measure the overhead of the server side daemons, we selected a number of popular internet daemons: the Apache HTTP server, the Nginx HTTP server, OpenSSH and ProFTPD. As with the previous test, the base run is performed by compiling the programs with CIL and subsequently running the programs. The measurements are relative to the CIL base run. To test the daemons, we transferred a 1.8 GB file via the tested service by connecting to the daemon via the local host. Before the start of each measurement, the disk cache was purged. Each service with and without mitigations was measured 50 times and the performance overhead for the averages of these runs are presented in Figure 4. The overheads range from -0.62% to 0.48% for both FreeSentry versions on these runs, which can be explained by regular variances in runtimes. This shows that running FreeSentry with disk intensive applications has no performance impact.

2) Opting out of protection for specific functions: As discussed in Section III-G, a programmer can decide to opt out of protection to improve performance. We tested the potential impact of this approach by profiling the *gap* program in the CPU2000 benchmarks. This program implements a language and library designed for computational group theory and is moderate in size, consisting of 71,430 lines of code. Profiling shows that *gap* spends most of its time in 2 functions when used with FreeSentry: *NewBag()* and *ProdInt()*. The *Newbag()* function allocates memory for a 'bag' of a particular type, sets the entries to 0 and returns a pointer to this memory. This is a memory allocation function that can reuse existing bags if needed and can provide garbage collection. This 150-line function runs on top of the regular memory allocation functions provided by *malloc()* and *free()* and calls them when new memory is needed or memory is released back to the system. The *ProdInt()* function is 174 lines and returns the product of 2 integers. For our test, we assumed that these functions were safe and opted out of protection for just these two functions, which means the pointers within these functions were not tracked. Note that for the *Newbag()* function, any memory that is allocated or freed will still be intercepted by our *malloc()* and *free()* functions, so memory will still be labeled and invalidated. As such, all other functions are still protected even if their memory was allocated or freed by an unprotected function. Opting out for these 324 lines out of a total of 71,430, dropped the performance overhead for *gap* from 104.28% to 33.42% and reduced the number of calls to our pointer tracking function from 2.3 billion calls to 1.3 billion calls. While programmer cooperation is not necessary to use our mitigation, if an assertion can be made that particular functions are bug free, this can vastly improve performance for our mitigation.

Table I. SECURITY EVALUATION AGAINST REAL WORLD VULNERABILITIES

CVE id	Affected Programs	Result
CVE-2003-0015	CVS <= 1.1.14	Protected
CVE-2004-0416	CVS 1.12.x-1.12.8 and 1.11.x-1.11.16	Protected
CVE-2007-1521	PHP before 4.4.7 and 5.x before 5.2.2	Protected
CVE-2007-1522	PHP 5.2.0 and 5.2.1	Protected
CVE-2007-1711	PHP 4.4.5 and 4.4.6	Protected

B. Security Evaluation

1) Perlbmk vulnerability: Perlbmk is one of the programs that makes up the SPEC CPU2000 benchmark suite. It is a stripped down version of the Perl interpreter that removes many operating system specific functions and aims to create a version of Perl aimed specifically at CPU benchmarking.

In the function *yy_lex()* in the file *toke.c*, the program implements lexical analysis for Perl programs. While running the benchmark's reference load, the program exhibits a use-after-free vulnerability.

Listing 4. Perlbmk use-after-free vulnerability

```
d = s;                                  2576
if (PL_lex_state == LEX_NORMAL)
  s = skipspace(s);                     2578

if (PL_lex_state == LEX_NORMAL
  && isSPACE(*d)) {                     2618
```

The relevant code snippets are duplicated in Listing 4. At line 2576, *d* aliases pointer *s*. Then at line 2578, the function *skipspace()* is called. This function will, depending on the program state, end up calling *realloc()* on the memory pointed to by *s* to increase the size to allow for a larger line of text to be read into memory. When the *realloc()* in question is called using the reference load provided by the SPEC CPU2000 benchmark, the memory will be allocated at a different location and the data will be copied. When this happens, all references to the memory that *s* refers to are invalidated by our mitigation. This includes the pointer *d*. When *d* is dereferenced at line 2617, the program crashes because it refers to freed memory. To be able to measure the performance overhead of *perlbmk*, we fixed the program using the least intrusive method by recording the value of *isSPACE(*d)* in a variable at line 2577 and then using this variable in the comparison at line 2617. Fixing this vulnerability resulted in no further crashes in the program.

2) Real world vulnerabilities: In this section we evaluate our mitigation against vulnerabilities that were found against real world applications. We selected the vulnerabilities solely on the availability of public proof-of-concept exploits (POCs). This limited our selection to 5 vulnerabilities.

The results of running these POCs against our mitigation are presented in Table I: a value of "protected" means that the attempted use-after-free was prevented and the program crashed trying to dereference an invalidated pointer (i.e. dereferencing a pointer to memory above 3GB in our implementation). The programs operated normally when not running the exploit. The focus in Table I is on double free vulnerabilities as these are the vulnerabilities for which public exploits are available. While many other use-after-free vulnerabilities have been discovered in C programs, there are very few with public exploits for open source C programs. For mitigations that provide a safer memory allocator, a double free vulnerability is the easiest type of vulnerability to detect: simply mark a chunk as free and check if it is free before performing a second free operation. However, for our mitigation with its focus on pointer tracking, double free vulnerabilities are the exact same thing as a use-after-free vulnerability as they both result from a dangling pointer reference. In the case of a double free, the dereference on the dangling pointer occurs during the free operation (where we do not perform any extra checks with respect to double frees), while in the general use-after-free case, the dereference occurs in another operation. Given that both types of vulnerabilities require dereferencing the pointer, the double frees tested here provide the exact same information about our mitigation as a more general use-after-free vulnerability.

Finally, a vulnerability [9] was privately reported by Damien Millescamps to the ClamAV team, which would trigger a use-after-free in version 0.98.4 of the software when scanning a maliciously crafted PE binary. The reporter provided a proof-of-concept binary that would trigger the vulnerability. Compiling ClamAV with FreeSentry resulted in the program crashing trying to access an invalid pointer as soon as the free memory is accessed, preventing exploitation of this vulnerability.

VI. Related work

A typical approach to preventing use-after-free vulnerabilities is to use garbage collection. However this does not deallocate memory instantaneously, but defers this to a scheduled time interval or till memory constraints require it to be collected. When garbage collection is done, only memory to which no references exist anymore is deallocated, preventing pointers from referring to deallocated memory [10]. However, C programs will generally not clear all pointers to a memory location when they free that location. As such, using garbage collection without modifying the program could result in the program using an unacceptable amount of memory. It also requires abandoning potentially customized memory allocators and using the garbage collector instead.

Dhurjati and Adve [11] propose a mitigation that protects against dangling pointers by ensuring that a new virtual page is used for every allocation of the program. To conserve memory this new virtual page is mapped onto the same physical page as the original allocation. This allows the mitigation to prevent stale pointers from using the memory, but also reduces memory overhead by reusing the original allocation. While the approach also has a low overhead for UNIX servers, it suffers from a significant slowdown when used with programs that perform frequent memory allocations.

Cling [12] is a memory allocator that is designed to prevent attacks against use-after-free vulnerabilities by making them harder to exploit. The overhead for the approach is very low, but some of the performance improvements and slowdowns are achieved due to the fact that a different memory allocator is used, which makes it harder to compare overhead completely. Cling does not eliminate the use-after-free problem entirely. It only allows address space reuse for freed objects among objects of the same type. However, while it constraints exploitation vectors, it does not really solve all issues introduced by dangling pointers if the program's control flow makes it hard to guess the object type being allocated. It also does not support protection against dangling pointers that refer to memory on the stack.

Undangle [13] is another mitigation that prevents use-after-free vulnerabilities. Like Cling, it does not require source code. It works by using taint tracking to track how pointers are copied to other pointers. When memory is destroyed, it can then see what pointers still point to the memory location. It can report all the dangling pointers that point to a particular memory location at a given set in time and allows a user to specify a window when to report these findings, allowing it to be used as a bug tracking tool, but which can result in false positives. Due to the very high performance overhead, the approach is not really practical for deployment use and relies on execution traces to perform analysis.

CETS [8] provides a compile-time approach to protect against dangling pointers in C. It does this by maintaining a unique identifier with each object and then associates the identifier with a pointer when the pointer is set to point to the object. Whenever a pointer is dereferenced, the mitigation checks if the pointer's unique identifier is still allocated. CETS achieves a mean overhead of 48% for the programs measured. However, due to a lack of robustness in their prototype implementation, a number of the more complex SPEC CPU benchmarks were not able to compile with their approach. If we compare the 14 benchmarks that overlap between both studies, the mean overhead for CETS for these benchmarks is around 48%, while FreeSentry has an overhead of around 29% with stack protection enabled and 12% with only dynamic memory protection enabled. CETS does not have the option of enabling only dynamic memory protection. Due to the lack of complex benchmarks supported by CETS it is also unclear if there would be an additional impact on performance for the regular benchmarks when adding support for these more complex programs.

DieHard [14] is a memory allocator that is designed to probabilistically tolerate errors including buffer overflows and dangling pointers. It does randomized allocation within a heap of a particular size, meaning that chunks of memory are allocated at random locations within this memory area. To prevent use-after-free attacks, it also randomizes the reuse of chunks. DieHarder [15] extends this approach by improving randomization which makes exploitation harder, but attackers who can control allocations (such as an attacker using JavaScript), could still simply allocate memory until their desired chunk is reused. DieHarder has a comparable geometric mean performance overhead to FreeSentry: DieHarder has an overhead of around 30% when compared to *dlmalloc 2.7*.

SAFEDISPATCH [16], VTGuard [17] and VTV [18] are three approaches that protect against the most widely used technique to exploit use-after-free vulnerabilities: overwriting virtual table pointers. These types of pointers are used in C++ objects to be able to support dynamic dispatching, so that the virtual method to execute for

the desired class can be decided at run-time. These approaches focus specifically on protecting these tables and thus their overhead is very low. However, if an attacker does not target the virtual table pointer, but instead targets a pointer within an object (such as the vulnerability discussed in Section II) then these mitigations would not be effective.

There are also a number of other approaches that combine bounds checking and dangling pointer mitigation.

Safe C [19] is a bounds checker for C that also provides protection against dangling pointers. It defines a kind of safe pointer that contains the following attributes: value, pointer base, size, storage class (heap, local, global) and capability (forever, never). The value attribute is the actual pointer, the base and size attributes are used for spatial check while the storage class and capability attributes are used for temporal checks. However the pointer representation is changed, resulting in an incompatibility with existing code. The added checks also have a significant impact on performance.

Clause et al. [20] developed a dynamic taint tool that checks for both spatial and temporal errors for dynamically allocated memory. It works by assigning taint marks to objects and assigning the same taint mark to pointers to these objects. The taint marks for pointers are then propagated and transformed through the program whenever an operation (such as arithmetic) on a pointer occurs. When the pointer is dereferenced, the taint mark for the pointer is compared to the taint mark of the object. If the taint marks differ then a memory error has occurred. The approach discussed by Clause at al. works on binaries rather than source code, but requires hardware assistance to be able to efficiently check and propagate the taint marks.

Fail-safe C [21] is a compiler that implements a memory safe version of the ANSI C standard. It does this using a number of techniques: fat pointers and integers (because pointers can be cast to integers and back again) for bounds checking, keeping track of runtime type information, garbage collection to prevent dangling pointers, etc. The overhead of this approach is, however, significant. The programs in the ByteMARK benchmark were slowed down by two to four times on average.

Xu et al. [22] track metadata that they associate with pointers to provide checks for both spatial and temporal errors and flags the spatial or bounds errors when a pointer is dereferenced. As with the previous approach, the overhead is significant, on average the benchmarks reported in the paper were slowed down up to two times.

Safe languages are languages where it is generally not possible for any known memory corruption vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. There are safe languages [23], [24], [25], [26], [27], [28] that remain as close to C or C++ as possible, these are generally referred to as safe dialects of C. While some safe languages [29] try to stay more compatible with existing C programs, use of these languages may not always be practical for existing applications due to the effort required to transform a project to adhere to the syntactical changes imposed by these languages.

VII. Future work

While this approach focuses specifically on preventing and measuring the technique to provide protection against use-after-free vulnerabilities, the approach can be extended to include bounds checking techniques that perform bounds checking using the available bounds information [3] for a modest increase in performance. While both the FreeSentry and bounds checking mitigations have non-trivial overhead, their combination should keep the overhead relatively close to the numbers of the worst performing technique, since processing for both techniques occurs during pointer creation and modification and not when the pointer is accessed.

VIII. Conclusion

There are many widely deployed mitigations, including stack cookies and address space layout randomization that are present in many of the current compilers and operating systems. However, there are currently no widely deployed mitigations that prevent use-after-free vulnerabilities. This has resulted in use-after-free vulnerability becoming the most exploited type of vulnerabilities on Windows operating systems. In this paper, we presented FreeSentry, a mitigation which is transparent to an unwitting programmer allowing easy deployment to provide protection, but also provides the flexibility to allow programmers who are aware of the mitigation to optimize their interaction with it. This allows us to offer a more focused protection, which allows a more complicated mitigation to be applied, with a lower measured overhead than the ones that are currently deployed. Our performance overhead is moderate for CPU-intensive

programs, while for programs that have high I/O it has no impact on performance, allowing it to be deployed as-is for server applications that rely on heavy I/O. In environments where security is of paramount importance, this type of mitigation can significantly improve security at a modest cost.

ACKNOWLEDGEMENTS

The author would like to thank Matthew Watchinski, Richard B. Johnson, David Suffling, David A. Raynor, Jason V. Miller, Miet Loubele, Aaron Adams and Donato Ferrante for their insightful comments during the development of the mitigation. The author would also like to thank Juan Caballero for his help as shepherd for this paper and the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against c and c++ programs," *ACM Computing Surveys*, vol. 44, no. 3, Jun. 2012.

[2] S. S. Nagaraju, C. Craioveanu, E. Florio, and M. Miller, "Software vulnerability exploitation trends"," Microsoft, Tech. Rep., 2013.

[3] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for C programs," in *ACM Symposium on Information, Computer and Communications Security*, April 2010.

[4] CVE-2009-0749, "Use-after-free vulnerability in the gifreadnextextension function in lib/pngxtern/gif/gifread.c in optipng 0.6.2."

[5] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of the Conference on Compiler Construction (CC'02)*, ser. Lecture Notes in Computer Science, vol. 2304, Grenoble, France, Mar. 2002, pp. 213–228.

[6] CVE-2010-0425, "Apache mod_isapi dangling pointer."

[7] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th USENIX Security Symposium*, Montreal, QC, Aug. 2009.

[8] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the International Conference on Memory Management (ISMM 2010)*, Jun. 2010.

[9] CVE-2014-9050, "Heap-based buffer overflow in the cli_scanpe function in libclamav/pe.c in clamav before 0.95.4 allows remote attackers to cause a denial of service (crash) via a crafted y0da crypter pe file."

[10] H. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software, Practice and Experience*, vol. 18, no. 9, pp. 807–820, September 1988.

[11] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *Proceedings of the International Conference on Dependable Systems and Networks*. Philadelphia, Pennsylvania,: IEEE Computer Society, 2006, pp. 269–280.

[12] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *Proceedings of the 19th USENIX Security Symposium*. USENIX Association, 2010.

[13] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM Press, 2012, pp. 133–143.

[14] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *Proceedings of the 2006 conference on Programming language design and implementation*. Ottawa, ON: ACM Press, 2006, pp. 158–168.

[15] G. Novak and E. D. Berger, "Dieharder: Securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[16] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing c++ virtual calls from memory corruption attacks," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[17] K. Johnson and M. Miller, "Exploit mitigation in Windows 8," in *Blackhat USA*, Las Vegas, NV, 2012.

[18] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, 2014.

[19] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the Conference on Programming Language Design and Implementation*, Orlando, FL, Jun. 1994, pp. 290–301.

[20] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the 22nd IEEE and ACM International Conference on Automated Software Engineering (ASE 2007)*, Atlanta, GA, Nov. 2007, pp. 284–292.

[21] Y. Oiwa, "Implementation of the memory-safe full ansi-c compiler," in *Proceedings of the Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun. 2009, pp. 259–269.

[22] W. Xu, D. C. DuVarney, and R. Sekar, "An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs," in *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, October 2004, pp. 117–126.

[23] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference*, Monterey, CA, Jun. 2002, pp. 275–288.

[24] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in Cyclone," in *Proceedings of the Conference on Programming Language Design and Implementation*, Berlin, Germany, Jun. 2002, pp. 282–293.

[25] G. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, Portland, OR, Jan. 2002, pp. 128–139.

[26] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, "Righting software," *IEEE Software*, vol. 21, no. 3, pp. 92–100, May 2004.

[27] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," in *Proceedings of the 2003 Conference on Language, Compiler, and Tool Support for Embedded Systems*, San Diego, CA, Jun. 2003, pp. 69–80.

[28] S. Kowshik, D. Dhurjati, and V. Adve, "Ensuring code safety without runtime checks for real-time control systems," in *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, Grenoble, France, Oct. 2002, pp. 288–297.

[29] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "CCured in the real world," in *Proceedings of the Conference on Programming Language Design and Implementation*, San Diego, CA, 2003, pp. 232–244.