Zero-sized heap allocations vulnerability analysis

Julien Vanegue
jvanegue@microsoft.com

Microsoft Security Engineering Center
One Microsoft Way, 27/1707
98052, Redmond, WA, USA.

Abstract

In this article, we discuss a source of security vulnerabilities related to zero-sized heap allocations. We present a feasibility study to show the use of a theorem prover based extended static checker to help code audit to find these vulnerabilities. We employed this tool to uncover around 10 local and remote untrusted code execution vulnerabilities in three core OS components. We highlight the benefits, the challenges faced and outstanding problems to enable wider use. Additional manual code review of remotely exposed software suggests that zero and near-zero allocations are particularly difficult to handle for developers.

1 Introduction

The dynamic memory allocator is a fundamental component of modern operating systems. Among all critical vulnerabilities found and fixed every year by software industry vendors, heap-based buffer overflows are one of the most important sources of security threats. Such defects can allow unauthorized users to elevate their credentials via untrusted code execution on vulnerable computers.

In this article, we show how formal methods can aid the detection of particular weaknesses of the heap management. The case of zero-sized heap allocations is not specific to any operating system and can affect both userland and kernel-land allocators. The article is divided into four main developments:

- We reveal the exact nature of the weakness introduced by zero-sized allocations and provide a taxonomy of all tested operating systems (both in the Windows and UNIX world). Most OS allocators are exposed as calling their API returns valid memory chunks when allocation functions are called with size 0. Returning NULL can also be a problem in some conditions that we detail below.
- We present our experiments in using an extended static checker HAVOC [1], a heap-aware verifier for C programs. We have deployed the analyser on multiple kernel components, some of them reaching one million lines of C code. The analyser produces a reasonable amount of warnings without any complex configuration.
- We present one real vulnerability uncovered in a core OS component. Other similar vulnerabilities were also found when multiple components are interacting.
- We show an alternative configuration of the tool suitable to detect vulnerabilities when the size of heap chunks is in the neighbourhood of zero (e.g. near-zero allocations) and give another uncovered remote vulnerability.

We want to emphasize that this weakness should not be considered as a new vulnerability class (such as buffer overflows), but rather a new type of code defect in the same style as integer overflows, as many occurrences are legitimate and do not lead to a vulnerability.

2 Zero allocation basics

Zero allocations are considered valid behaviours of programs as defined by the *ISO C99* standard and its ancestors. In essence, the standard is described with the following words: *"If the size of the space requested is zero, the behaviour is implementation defined : either a* NULL *pointer is returned, or the behaviour is as if the size were some non-zero value, except that the returned pointer shall not be used to access an object."*.

```
 1. void       f()
 2. {
 3.  PSTRUCT  data;
 4.  UINT     size;
 5.  PSTRUCT2 ptr;

 6.  data = readstruct();
 7.  if (data->nbr > MAXSHORT)
 8.    return ERR;
 9.  size = data->nbr * 2;
10.  ptr = Alloc(size);
11.  if (ptr == NULL)
12.    return ERR;
13.  ptr->field = data->field;
    (...)
```

```
1. void g(char *buf)
2. {
3.   UINT size;
4.   PUCHAR pstr;


5.   size = readint() + sizeof(TYPE);
6.   ptr = Alloc(size);
7.   if (ptr == NULL)
8.     return -ERR;
9.   memcpy(ptr, (PTYPE) buf, sizeof(TYPE));
     (...)
```

Figure 1: Zero allocations vulnerabilities

Figure 1 shows two different zero allocation vulnerabilities in independent functions *f* and *g*. First in *f* : an untrusted structure value of type *STRUCT* is written to variable *data* on line 6. Later on line 9, a size variable is computed from the value of the unknown value held in the `data->nbr` untrusted structure field. Note that the untrusted value is smaller than *MAXSHORT* (a constant defined to *0xFFFF* on most systems) since line 7 ensures that bigger values are discarded, as to avoid any integer overflow on variable *size*. However, if `data->nbr` has value 0, the *Alloc* function (either *malloc*, or *kmalloc*, etc) can either return an error value *NULL* (well checked in example 1), or the address of a valid heap chunk (as implemented by most allocators) allocated with only zero bytes of size (in reality, a few padding bytes are always allocated). A latter memory write happens when the `ptr` variable is dereferenced, and written to out of bounds. A successful attack requires that the offset of the accessed structure field is bigger than the allocated memory size on line 10. This condition will often be satisfied in the vulnerabilities we have found.

Another zero allocation vulnerability can arise when an untrusted value is manipulated during integer arithmetic. In vulnerable function *g*, a `size` variable is computed from an untrusted value returned by the *readint()* function. A constant type header size is then added on the total allocated size. As this operation happens on computers with modulo arithmetic (and not on infinite integers), the size value can go beyond the $2^{32} - 1$ limit and become a small number. A buffer overflow happens on line 9 as the allocated size is smaller than the size of *TYPE*. Note that we can characterize this second example as a zero allocation vulnerability since the value returned by *readint()* could be anything, including the value that reset variable `size` to 0.

We do not consider zero allocations to be a new class of vulnerabilities. Indeed, allocating zero bytes is a safe practice as long as proper handling of the allocated buffer is performed before and after the call to the dynamic allocation function. Additionally, sanitization must be performed on untrusted variable components present in the `size` variable, as to avoid any boundary condition that would lead to memory safety issues. Zero allocations are one instance of such boundary value, but similar analysis can be performed on other boundary values as to uncover corner cases that are harder to handle for software developers.

3 OS allocators classification

The detection of zero allocation vulnerabilities is relevant on all operating system. We summarize this information in the table of figure 2. There are two possible exposed behaviour of the heap allocator concerning zero-sized requests:

- Either the allocator returns a valid chunk address. Writing to the address can lead to heap corruption. The address generally points to a chunk of a few bytes (0 + small padding). The allocator will usually revert to a 16 bytes chunk as in most case, this is the smallest available size available cached chunk lists (either bins, slabs, slubs, etc). This is the most commonly encountered behaviour.

- Or the allocator function returns *NULL*. This is safe with the appropriate check for *retptr != NULL*, which can be forgotten in some cases (especially in kernel mode). Solaris OS behaviour in kernel mode allocators can be to return a *NULL* pointer. The default Linux kernel allocator will return a constant *0x10*, which bypass the traditional check to NULL and is also exposed to NULL page dereference type of attack. Of course, such attack will not succeed

	User-land	Kernel-land
Windows (7)	Yes	Yes
Linux 2,6 (deb)	Yes	Yes
Linux 2.6 PaX	Yes	No
FreeBSD (5.5)	No	Yes
NetBSD (3.0.1)	Yes	Yes
OpenBSD (4.4)	Yes	Yes
Solaris (Open/10)	Yes	Yes
Mac OSX (Leopard)	Yes	Yes

Figure 2: Exposed memory allocators

if NULL page protection is properly implemented, which we witnessed is not always the case [22].

In all occasions but two, we found that the tested allocator was unsafely exposed to zero allocation vulnerabilities. The non-exposed cases were found on FreeBSD user-land allocator and the Linux/PaX kernel memory allocator. On the tested FreeBSD default user-land allocator, passing zero to the *malloc* function will return a constant *0x800*. This value happens to be a non-mapped address. Unless a new intrusion technique consisting of mapping the NULL pages of privileged binaries comes out, we can assume this behaviour is safe from attacks. When applying the PaX patch on the Linux kernel, the return value of zero allocation calls is changed for *0xFFFFF000*, which is the address of an unmapped (in fact, the last) page in the kernel virtual address space. This modification gives a good mitigation for zero kernel allocations on Linux/PaX machines as it limits the impact of the vulnerability to a denial of service, effectively enforcing the *ISO* specification.

Given the wide exposed attack surface for this property, we investigated the automated static analysis of programs in order to find all such vulnerabilities automatically in large depot of C source code. We now explain how programs can be verified for the absence of zero allocations by a technique issued from theorem proving.

4 Extended static checking

A regular security analysis process involves a lot of manual code review. The lack of competent human analyst can be a motive why automated vulnerability analysis is desired. Moreover, the human error can lead to missing bugs. Nevertheless, automated tools also leverage the analyst's domain specific knowledge by making it possible to tweak the verifier's configuration or directives as to reflect implicit coding rules or guessed logical invariants that can be tricky to find automatically.

There are two main techniques of automated vulnerability analysis currently being deployed in the computer security industry. The most successful techniques are called fuzz testing and static analysis. The two techniques are complementary, as fuzz testing allows non-exhaustive deep traces visits in the test space, while static analysis is fully exhaustive but limited to a single module and pre-defined properties. The two techniques are able to uncover zero allocation bugs. We will focus on static analysis in the remaining of this article. Static analysis is a pure *white box* technique where plain source code (or clear-text binary code) is required. Static analysers can be configured to focus on the most dangerous programming problems such as buffer overflows, as well as other classes of vulnerabilities known to lead to an unauthorized code execution. To the authors' knowledge, there is no equivalent of such targeted analysis in the fuzz testing world. Unlike fuzz testing, one powerful machine is sufficient to perform full coverage analysis on a large depot of code. However, static analysis is usually bounded to a single program or OS component. The combination of a simple expression of a security property with the application of a very precise analysis tool is a key element of our experiment. We can discard false positives by a manual review of warnings. Fuzz testing often offers a quicker reward when you do not have to find all vulnerabilities. However, the cost of triaging critical from benign vulnerabilities found via fuzzing can also be time-consuming. In the case where missing vulnerabilities is not an option, static analysis can assure a complete coverage of the analysed code. This is possible because the analyser considers a superset of the analysed program's behaviours as to avoid missing any corner case that can be forgotten by dynamic testing. One can also create unsound static analysis tools that will not find all vulnerabilities but will keep the false positive rate near zero.

4.1 Theorem proving aided code review

We focus on the use of a static verification technique called theorem proving. This technique makes it possible to analyse each program path separately as to avoid introducing any approximation in summaries at program merge point. Theorem proving theory for call-free, loop-free programs is pretty well established and implemented in many modern tools [28] [39] [29] [30]. Our experiments on theorem proving C programs for the discovery of zero allocation vulnerabilities relied on using HAVOC/Boogie/Z3, a collection of formal analysis tools developed in Microsoft Research. HAVOC is a heap-aware verifier for C programs [1] that models the semantic of C constructs (including pointer manipulation) precisely and accurately. HAVOC relies on the Boogie theorem prover [24] to construct the verification condi-

```
void func(int x, int b)
{
  if (x > 0)
   {
    if (b > 0)
      y = x + b;
    else
      y = x - b;
   }
  else
    y = -x + 1;
  // precond: requires(y != 0)
  malloc(y);
}
```

```
void func(int x0, int b0)
{
  // if (x0 > 0)
      G1 = (x0 > 0);
      // if (b0 > 0)
          G2 = (b0 > 0);
          y1 = x0 + b0;
      //else
          y2 = x0 - b0;
     //endif
      y3 = (G2 ? y1 : y2);
  //else
      y4 = -x0 + 1;
  y5 = (G1 ? y3 : y4);
  assert(y5 != 0);
}
```

```
// Verification condition
IsSatisfiable(
    y1 = x0 + b0
  ∧ G1 = (x0 > 0)
  ∧ G2 = (b0 > 0)
  ∧ y2 = x0 - b0
  ∧ y3 = (G2 ? y1 : y2)
  ∧ y4 = -x0  + 1
  ∧ y5 = (G1 ? y3 : y4)
  ∧ ¬ (y5 != 0)
);
```

Figure 3: Construction of the verification condition on C code via SSA form

tion. Boogie constructs a first-order logic formula that represents the program execution. HAVOC allows user-defined annotations to be part of the verification condition. When Boogie calls the constraint solver (in our case, Z3 [25]), the verification condition is either proved or violated. A violation reveals that the program specification (reflected by code annotations) is not respected, and we may have found a vulnerability. HAVOC is very attractive for the security analyst as it makes it possible to quickly capture domain specific knowledge of the analysis targets by writing code annotations. The annotation language recognized by HAVOC is well documented and usable through a Microsoft C compiler plug-in. HAVOC analyses real C code and not a subset of C. Real C programs can be analysed but C++ support is still work in progress. Bit vectors arithmetic can also be enabled as to gain additional granularity on the modelling of arithmetic operations, but this feature still has to be experimented on a larger scale. We give an example of sequences of logical operations realized by the verifier using a simplistic loop-free example in Figure 3. We assume that the reader is not familiar with theorem proving and this example is chosen very simple on purpose. The analysis is two fold. First, transformation into the *Static Single Assignment* (SSA) form [45] allows the disambiguation between variable versions across different program paths. Construction of the verification condition from SSA form is straightforward as it consists of taking the conjunction of every SSA program statements. Note how the precondition formula for *malloc* is negated in the VC as to make the whole formula false if such invariant does not hold. The analyser handles loops conservatively : if a value of interest (e.g. the allocation size variable) is modified in the loop, HAVOC will report that the variable can have any value. It is then possible to enforce a specific loop invariant to make this warning go away. Fortunately, the number of loops in a program is generally fairly small (even in large code bases) and only those loops whose manipulated variables have an incidence on the size of an allocation are considered, avoiding any unnecessary annotation burden. Warnings produced by such analysis are generally fast to review : only few days are necessary to review all warnings in one million lines of C code. The review of alarms can be time-consuming if warnings appear in functions that have lots of callers. An iterative static analysis process is then used to refine the warnings list. At some point, loop invariants are the only remaining annotations that can eliminate the last spurious warnings. Adding annotations and running the analyser avoids manual code review where it can be proven that the program behaves as expected. An extension of the HAVOC tool allows inter-procedural analysis based on a fixed-point algorithm called *HOUDINI* [44]. We have not made use it this feature and the remaining of the article focuses on results obtained by intra-procedural analysis only.

5 Detecting complex vulnerabilities

In this section, we justify our approach of detecting the zero allocation problems at the allocation sites rather than at the corrupted memory access site. The code snippets we used so far to illustrate the zero allocation problem have been reasonably simple as the zero allocation and the memory corruption were happening in the same function. Many times, the corruption involves independent functions in the analysed module (by *independent*, we mean that each function is not either a caller or a callee of the other one. More precisely, none of the function is in the cone of the other on the call graph). We call those problem *multi-trace vulnerabilities*. In that case, the standard forward inter-procedural analysis is unable to detect a memory corruption as the zero allocation and the memory access happens on independent traces of the call graph. Sometimes, functions are even located in independent modules. In those cases, accurate and automated detection of sparse vulnerabilities is very expensive as the static analysis must cross module boundaries. We give examples of real vulnerabilities matching those criteria that we detected by enforcing a precondition on the allocation function rather than on the memory dereference site. This choice introduces conservative approximation into our analysis but allows us to find complex problems that would require more computational power and implementation effort to be answered without any false positives.

5.1 Multi-trace vulnerabilities

Vulnerabilities involving multiple functions are common. Inter-procedural analysis allows to uncover such problems when the whole vulnerability can be characterized on a single program trace. In that case, the analysis starts from a program entry point and goes deeper into the analysed components as functions are called from the entry point, and so on. In some cases however, a state corruption (involving a global variable) can happen on one trace but the real memory corruption happens somewhere else as an indirect consequence of the state corruption. This scenario is shown in Figure 4. First, a zero allocation happens on line 6 of function *_SetData* as a result of a non-sanitized multiplication on line 9 of function *Syscall*, an entry point of the kernel. No further memory manipulation happens on this code path. On another code trace starting with function *Syscall2*, the kernel will lookup the zero allocated buffer and dereference it (line 9), leading to a buffer out of bound access. No code execution is possible here as the invalid dereference happens on the right hand side of the expression. This example is only used to show a concrete case where a multi-trace vulnerability can happen. This code problem is correctly detected by our tool thanks to the approximate characterization of this vulnerability class of using preconditions on the allocation function.

5.2 Inter-module vulnerabilities

When analysing large programs composed of many modules and implemented over many millions of lines of code, useful program invariants may be hard to infer as many logical conditions are out of reach. This happens when the program events that trigger the complete zero allocation vulnerability are distributed across different parts of the analysed program or operating system.

```
1. NTSTATUS Syscall(HANDLE h, PSTRUCT1 pData)
2. {
3.     PSTRUCT1  safedata = Handle2Ptr(h);
4.     DWORD     count;
5.
6.     try {
7.       STRUCT2 copy = ProbeAndRead(pData);
8.       if (copy.flags & COND_FLAG)
9.          count = (copy.field1 * sizeof(HDR))
10.               + (copy.field2 * sizeof(DWORD));
11.     } except { return ERR; }
12.     return (_SetData(safedata, &copy, count));
13. }

1.  NTSTATUS _SetData(PSTRUCT1 safe,
                      PSTRUCT2 cur,
                      DWORD cnt)
2.  {
3.    PSTRUCT2 tmparray;
4.
5.    if (cur->flags & FLAG_ENABLED) {
6.       tmparray = UserAllocPool(cnt, TAGS);
7.       if (tmparray == NULL) return ERR;
8.       safe->array = tmparray + sizeof(HDR);
9.       try { memcpy(safe->array,
10.                   cur->array, cnt); }
11.      except { return ERR; }
12.   }
13.  return ESUCCESS;
14. }

1.  NTSTATUS Syscall2(HANDLE h)
2.  {
3.     PSTRUCT1 p1;
4.     PSTRUCT2 p2;
5.
6.     if (p1->flags & FLAG_ENABLE) {
7.       p1 = Handle2Ptr(h);
8.       if (p1->array == NULL) return ERR;
9.         p2 = p1->array[p1->field2];
10.    }
11.  return ESUCCESS;
12. }
```

Figure 4: A multi-trace vulnerability

Figure 5 shows an example of such real-world complex vulnerabilities that our review was able to uncover. In the case of multi-components vulnerabilities, we cannot afford proving invariants across different modules implemented in many millions of lines of code. Interprocedural inference is likely to converge very slowly in those cases as complex cycles can happen in the call graph. Moreover, we want to keep our analysis compositional across operating system modules, so that global safety can be proved by analysing components independently. Figure 3 shows a zero-size heap allocation vulnerability taken from a real case that happened in the operating system. In this case, the malicious remote user could control the size of an allocation by making specific request on a server driver. The driver then forwards the request to the corresponding file-system driver through the *IO manager*.

The *IO manager* is a central component in the *Windows* kernel as it controls the creation and forwarding of interrupt requests between drivers. OS knowledge furnished by the analyst (such as preconditions for *IoDoRequest* as indicated on Figure 5) allows enforcing specific conditional preconditions on components interface functions. This makes it possible to catch inconsistencies when they cross module boundaries, without having to analyse deeper in the system, as to avoid a complexity blow-up.

6 Results

This section sums up our experiment results. The table in figure 6 contains measurements from analysis experiments for some core parts of the operating system. Those numbers are specific to the zero allocation property and would be different for other properties or analysed components.

COMPONENTS	A	B	C	All
LOC	1M	100K	200K	1.3M
Checked assertions	618	161	15	794
Warnings w/o annots	101	40	7	148
Found vulnerabilities	5	1	3	9

Figure 6: Analysis results on components A, B and C

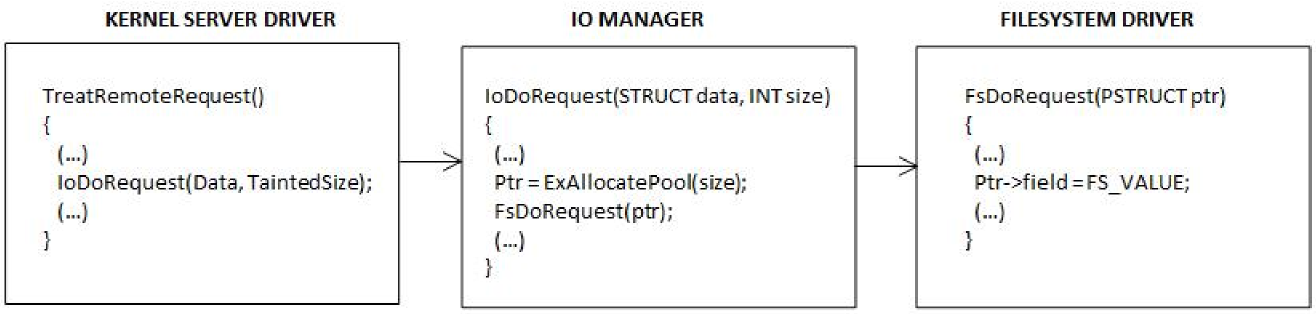
The initial list of warnings contains a high amount of false alarms. This is not because of infeasible paths that is considered vulnerable, as HAVOC automatically discards such spurious traces. Remaining cases are those that cannot be resolved only with intra-procedural analysis. We eliminated all other false alarms by a simple manual inspection. For some warnings that were too time-consuming to review, we added refined precondition on functions containing the alarm as to enforce a specific precondition. This has for effect to remove the original warning but make appear other new ones if the precondition is violated in any of the caller contexts. Most of those false alarms were eliminated by adding annotations on few layers of functions. After a few iterations of this process, a very small number of alarms remain, and we review them in deep details. We managed to analyse a million lines sized component (such as *A*) and uncover multiple issues at a fairly successful rate. The tool was most effective when focused on specific API with a history of security vulnerabilities, so that the tool can be targeted precisely without much noise as for component *C*. Overall, HAVOC provided a configurable, transparent, and high-coverage solution for the analysis of such security property.

```
1. __kernel_entry
2. NTSTATUS       syscall(PVOID pParam)
3. {
4.  PSETTING      pitem;
5.  PLARGESTR     pls;
6.  UINT          cnt;
6.  if (pParam) {
7.    try {
8.      pitem = (PSETTING)pParam;
9.      cnt = ProbeAndReadUlong(&pitem->size) + 1;
10.     pls = UserAllocPool(cnt);
11.     if (pls == NULL)
12.       return -ERR;
13.     memcpy(pls, pParam, cnt);
14.   }
(...)
15.   return _InternalFunc(NULL, pls);
16. }

1. NTSTATUS _InternalFunc(PVOID p, PLARGESTR pls)
2. {
3.  LARGESTR str;
4.  if (p == NULL)  {
5.    str.buff = UserAllocPool(pls->Length + 1);
6.    if (str.buff == NULL) return -ERR;
7.    try {
8.       str.Length = pls->Length;
9.       memcpy(str.buff, pls->buff, str.Length);
10.      str.buff[str.Length] = 0;
(...)
```

Figure 7: Kernel zero allocation vulnerability

One fixed elevation of privilege vulnerability in the kernel is shown in Figure 7. In this example, the code execution flows from function *syscall* (one of the kernel entry point) to an internal kernel function. An intermediate large string is allocated with size value passed from user-mode via the `pParam` parameter. As the parameter is casted to a *PSETTING* pointer type, one of its field is validated via the *ProbeAndReadUlong* kernel macro that ensures the validity of this pointer variable. This check is indeed necessary as to forbid the user-mode parameter to point on a kernel memory address and potentially lead to further kernel corruption. *Probe* functions triggers an exception if the supplied pointer is invalid, which is why a *try* block is necessary when reading the data. Additionally, a data copy is performed by the *Probe* call so that no time-of-check / time-of-use concurrency vulnerability can happen. A problem happens because the value pointed by this valid address is used without any bound checking, allowing an integer overflow to happen on the `count` variable. The direct consequence of our computers' modulo arithmetic is to trigger a zero-sized allocation, that is not caught by error checking at line 11 since the call returns a valid heap chunk pointer. A memory copy then happens with a zero byte size, which is a no op. Then the allocated



```
TreatRemoteRequest()
{
  (...)
  IoDoRequest(Data, TaintedSize);
  (...)
}
```

```
IoDoRequest(STRUCT data, INT size)
{
  (...)
  Ptr = ExAllocatePool(size);
  FsDoRequest(ptr);
  (...)
}
```

```
FsDoRequest(PSTRUCT ptr)
{
  (...)
  Ptr->field = FS_VALUE;
  (...)
}
```

Figure 5: Inter-module zero allocation vulnerability

pointer is passed to the internal function. There, a new *LARGESTR* object is created whose buffer is allocated with an unknown size (as variable *pls* points on an uninitialized memory chunk). In practice, an attacker may use a heap spraying attack as to place untrusted data in the next contiguous heap chunk. A memory copy of uncontrolled size can then corrupt the heap with untrusted data. This type of vulnerability generally get patched with a higher priority as they can allow untrusted code execution.

Other vulnerabilities can happen when the characterizing class is slightly different than a zero allocation. We make explicit such limitation of the presented analysis in the following section about near-zero allocations vulnerabilities.

7 Near-zero allocations vulnerabilities

The previous analysis is only suitable to detect vulnerabilities when the allocation size is exactly zero. Sometimes however, security problems arise when the allocation size is not zero but in the neighbourhood of zero. Such problem can be found when the size variable is incorrectly handled even though no zero allocation can happen. We found several of such problems in the presence of additional code defects. Let us show one remote vulnerability that we found by code review of allocation sites in a user-land client software. In this case, the allocation size is bigger than (but in the neighbourhood of) zero, as a constant small amount of padding is always added to the size before the call to the allocation function. However, when the size variable does not account for this padding, a de-synchronization happens that is naturally more difficult to handle for developers.

The code in figure 8 is taken from a network client

```
 1. BOOL DecodeStr(PCHAR input, UINT sz, PSTAT st)
 2. {
 3.   st->newcnt = GetFieldLen(input, sz);
 4.   st->tmpbuf = malloc(st->newcnt + 2);
 5.   if (NULL == st->tmpbuf)
 6.     return EOOMEM;
 7.   memcpy(st->tmpbuf, input, st->newcnt);
 8.   return (HdrStringConvert(st));
 9.}

 1. BOOL HdrStringConvert(PSTAT st)
 2. {
 3.   if (NULL == st->finalbuf) {
 4.       if (0 == st->newcnt) return ERR;
 5.       st->finalbuf = calloc(st->newcnt);
 6.       if (NULL == st->finalbuf)
 7.          return EOOMEM;
 8.    } else {
 9.       UINT off = strlen(st->finalbuf);
10.       st->finalbuf = realloc(st->finalbuf,
                                st->newcnt +
                                off + 1);
11.       if (NULL == st->finalbuf)
12.          return EOOMEM;
13.       *(st->finalbuf + off) = SEP_CHAR;
14.       off++;
15.    }
16.   switch (st->tmpbuff[0] & 0x7F) {
17.     case ASCII:
18.     ConvertFromASCII(st->tmpbuff + 1,
                         st->newcnt - 1,
                         st->finalbuf + off);
(...)
```

Figure 8: Near-zero allocation vulnerability

software manipulating untrusted data. On line 3 in function *DecodeStr*, a length value field is read from a network input and stored in 16 bits variable `st->newcnt` . The function then allocates the corresponding memory size, adding two bytes. Note that no integer overflow can happens as the addition operation happens on 32 bits even if `st->newcnt` is on 16 bits.

The newly allocated string is then passed to function *HdrStringConvert* for conversion into a suitable format for internal storage and manipulation. This second function operates differently depending on the state of the `st->finalbuff` variable. Indeed, multiple calls to *DecodeStr* in a row will make the newly received string concatenated to the previously converted one, separating the two by a special character *SEP_CHAR* . In the case where `st->finalbuf` is *NULL*, the new count variable is properly checked for zero and an error is returned in that case. However, when the final buffer is not empty, a reallocation will happen without checking the value of the new counter. If the `st->newcnt` is zero, an integer underflow will happen on line 18 when the variable is decremented (as to reflect the special first byte of the string that contains the string encoding information). A buffer overflow happens in *ConvertFromASCII* function as a big amount of data is written out of bounds of the final buffer. Exploitation of this vulnerability depends on whether an attacker is able to control the uninitialized data contained in the memory chunk allocated at line 4 of function *DecodeStr*. Since the vulnerable code is located in a network handling code and that no memory zeroing happens when freeing heap memory chunks, conditions are likely to be reunited to trigger unauthorized remote code execution.

It is harder for an automated analysis to find such vulnerabilities as the allocation size is not exactly zero (since it is padded with a constant size value 2, making the required safety precondition always true). Additionally, a reentrancy analysis is required to reach the vulnerable context in function *HdrStringConvert*. Such characteristics explain why a simple zero check is insufficient to uncover those subtle vulnerabilities.

8 Related work

There are two distinct areas of related work. The first one is from the software verification community. A number of formal verification tools has been successfully deployed in the software industry. Either based on type qualifiers checking [43] [33], model-checking [27], data-flow analysis [31] [32] or fuzz testing [3], they found dozens of vulnerabilities, many of those are not specifically targeted on security relevant properties. Theorem proving has been used on source code to ensure user/kernel pointer validation [26], correct locking schemes [28] or the absence of **NULL** pointer dereferences [39]. Other experiments were made to detect integer overflows [40] [41]. Many more machine code analysis frameworks (either based on theorem proving [34] or data-flow analysis [38] [37] [36] [42]) are still at early stages of experimentation on large industrial software.

The second group of related works is from the offensive and defensive software technologies (known as *exploit* development and *mitigation*) community. Since initial publications of heap exploitation techniques [8] [9], a new generation of heap-aware security practitioners has extensively researched the topic of untrusted code execution in the presence of user-land heap management defects [10] [11] [12]. Pioneer address space layout randomization (ASLR) and non-execution protections [2] made heap exploitation more challenging. Allocator implementations were enforced by adding consistency checks during *chunk unlinking* operations, thus breaking known exploitation vectors. A second generation of heap exploitation techniques was developed [14] [15] [19] [17] targeting specific weaknesses that were left unchecked in new allocators. Many more techniques were published on kernel allocator attacks [16] [20] [21] on various OS. Modern non-execution protections can now benefit from hardware-level support [9], lowering the performance impact induced by software-level protections. As cloud-computing is emerging and web browsers are more exposed targets, new advanced exploitation techniques now make use of fine grained heap spraying in JavaScript [18] and ActionScript JIT compiler predictable behaviours [23] to take advantage of memory bugs even in the presence of advanced protections such as *DEP* and *ASLR*.

9 Conclusion

We uncovered new vulnerabilities in large and complex low-level software by performing a straightforward but systematic zero value analysis of specific operating system functions parameters. Using the HAVOC analyser made it possible to filter a major part of the problem space while keeping a small rate of false positives compared to path-insensitive implementations of formal verification. Such methodology is directly actionable during the industrial software development life-cycle and allows keeping a higher assurance of the absence of such problems in the future.

The author would like to thank Shuvendu Lahiri for his guidance on using HAVOC and writing this article. Thanks also to Mark Wodrich, Thomas Garnier, Matt Miller, and Ken Johnson for their feedback and assistance on security vulnerability research in Microsoft.

References

[1] HAVOC : Heap Aware Verifier for C programs
S. K. Lahiri, S.Qadeer.

[2] The PaX project.
The PaX team.
`http://pax.grsecurity.net`

[3] Automated whitebox fuzz testing
P.Godefroid, M.Levin, D.Molnar in Proceeding of the Network Distributed Security Symposium (NDSS'08)

[4] Data Execution Prevention
Microsoft TechNet portal, 2004.
`http://technet.microsoft.com/en-us/library/bb457155.aspx`

[5] OpenBSD execution protection
The OpenBSD project 3.3, 2003.
`http://en.wikipedia.org/wiki/W^X`

[6] INTEL processors XD (Execute Disable) bit protection
`http://www.intel.com/technology/xdbit/index.htm`

[7] New Security Enhancements in Red Hat Enterprise Linux v.3, 2004
The Red Hat project.
`http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`

[8] w00w00 on heap overflows
M.Conover, w00w00 security team website, 1999.

[9] JPEG COM Marker Processing Vulnerability in Netscape Browsers, Alexander Peslyak.
Solar Designer's bugtraq post.
`http://www.openwall.com/advisories/OW-002-netscape-jpeg/`, 2000.

[10] VuDo malloc tricks
M.Kampf, Phrack magazine volume 57, 2001.
`http://www.phrack.com/issues.html?issue=57&id=8`

[11] Once upon a free() ...
Scut, TESO, Phrack magazine issue 57, 2001.
`http://www.phrack.org/issues.html?issue=57&id=9#article`.

[12] Third generation exploitation (smashing the heap under win2k)
Halvar Flake, Blackhat Briefings 2002.
`http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt`

[13] Advanced Doug-Lea's malloc exploits.
JP, Core ST, Phrack magazine issue 61, 2003.
`http://www.phrack.org/issues.html?issue=61&id=6#article`, 2003.

[14] Reliable Windows Heap Exploitation (Win2KSPO through WinXPSP2)
M.Conover, O.Horovitz, CanSecWest conference 2004.
`http://cansecwest.com/`

[15] Malloc Maleficarum
Phantasmal Phantasmagoria, bugtraq post, 2005.
`http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt`

[16] Attacking the core (kernel exploitation notes)
E. Perla and M.Oldani, Phrack magazine issue 64, 2007.
`http://www.phrack.org/issues.html?issue=64&id=6#article`

[17] The use of set_head() to defeat the wilderness
J.S.Guay-Reloux, Phrack magazine issue 64, 2007.
`http://www.phrack.org/issues.html?issue=64&id=9#article`

[18] Heap Feng Shui in Javascript
A.Sotirov, Blackhat Briefings 2007.
`http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf`

[19] Yet another free() exploitation technique
Huku, GRHack, Phrack magazine issue 66, 2009.
`http://www.phrack.org/issues.html?issue=66&id=6#article`

[20] Exploiting UMA, FreeBSD's kernel memory allocator
Argp and Karl, Phrack magazine issue 66, 2009.
`http://www.phrack.org/issues.html?issue=66&id=8#article`

[21] Analyzing local privilege escalations in win32k
Thomas Garnier, uninformed journal volume 10, 2008.
`http://www.uninformed.org/?v=10&a=2&t=sumry`

[22] Bypassing Linux NULL pointer dereference exploit prevention, 2009.
J.Tinnes, T.Ormandis, blog post.
`http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html`

[23] Interpreter Exploitation: Pointer Inference and JIT Spraying
Dionysus Blazakis, Blackhat briefings DC, 2010.
`http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf`

[24] Boogie: A modular reusable verifier for object-oriented programs
M Barnett, BY Chang, R DeLine, B Jacobs, K.R.M.Leino in Formal Methods for Components and Objects (FMCO'05).

[25] Z3: An Efficient SMT Solver.
L.De Moura, N.Bjorner in Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08).

[26] Verifying the safety of user pointer dereferences
S.Bugrara and A.Aiken, in IEEE Symposium on Security and Privacy (SSP'08)

[27] The SLAM project: debugging system software via static analysis
T.Ball and S.K.Rajamani, in Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)

[28] Scalable Error Detection using Boolean Satisfiability.
Y.Xie, A.Aiken, Principles of Programming Languages (POPL'05).

[29] PVS: Combining specification, proof checking, and model checking
S. Owre, S. Rajan, J.M.Rushby , N.Shankar and M.Srivas, Computer Aided Verification, 2006.

[30] Isabelle: a generic theorem prover
L.C.Paulson, 1994 (Book).

[31] A static analyzer for finding dynamic programming errors
W.R.Bush, J.D.Pincus and D.J.Sielaff, Software Practice and Experience, 2000.

[32] Software validation via scalable path-sensitive value flow analysis
N.Dor, S.Adams, M.Das, Z.Yang - ACM SIGSOFT 2004.

[33] Finding User/Kernel Pointer Bugs With Type Inference.
R.Johnson, D.Wagner, USENIX Security 2004.

[34] BitBlaze: A New Approach to Computer Security via Binary Analysis.
Dawn Song and al, Springer-Verlag Berlin Heidelberg 2008.

[35] The ASTREE Analyzer.
P.Cousot and al.
`http://www.astree.ens.fr/`

[36] The ERESI project
The ERESI team
`http://www.eresi-project.org`

[37] REIL: A platform-independent intermediate representation of disassembled code for static code analysis
T Dullien, S Porst, CanSecWest conference 2009.

[38] CodeSurfer/x86a platform for analyzing x86 executables.

G.Balakrishnan, R. Gruian, T.Reps and T.Teitelbaum in Springer Berlin / Heidelberg Volume 3443, 2005.

[39] Calysto: Scalable and Precise Extended Static Checking
D.Babic and A.J.Hu, ICSE 2008.

[40] IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution
T.Wang, T.Wei, Z.Lin, W.Zou, Network Distributed Security Symposium, 2009.

[41] UQBTng: a tool capable of automatically finding integer overflows in Win32 binaries.
R.Wojtczuk, 22nd CCC Congress, 2005.

[42] Finding use-after-free bugs with static analysis.
S.Hernan's blog.
`http://seanhn.wordpress.com/2009/11/30/finding-bugs-with-static-analysis/`

[43] Flow-sensitive type qualifiers.
J.S.Foster, T.Terauchi, and A.Aiken in PLDI'02.

[44] Houdini, an annotation assistant for ESC/Java
C Flanagan, K Leino - FME, 2001.

[45] Efficiently computing static single assignment form and the control dependence graph
R.Cytron an al. in ACM Transactions on Programming Languages and Systems (TOPLAS), 1991.