Technical Disclosure Commons

Defensive Publications Series

December 2020

Detecting Dangling Pointers Using Embedded Metadata

Kentaro Hara

Benoît Lizé

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Hara, Kentaro and Lizé, Benoît, "Detecting Dangling Pointers Using Embedded Metadata", Technical Disclosure Commons, (December 30, 2020)
https://www.tdcommons.org/dpubs_series/3931

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Detecting Dangling Pointers Using Embedded Metadata

ABSTRACT

When the memory for an object is deallocated, pointers to that object become invalid. Such pointers, known as dangling pointers, can be exploited by attackers to cause undesirable or malicious program behavior. A pointer that points to memory that has been reused (reallocated) is known as a use-after-free pointer; these, too, are gateways to exploits. This disclosure describes techniques to detect the presence of dangling and use-after-free pointers in runtime and, upon detection, forestall security breaches due to such pointers by causing a program crash accompanied by a bug report. Per the techniques, both pointer and pointed-to object are augmented with metadata that enables, upon dereferencing, the checking of the validity of the pointer and the re-use status of the memory it points to.

KEYWORDS

- Dangling pointer
- Use-after-free memory
- Use-after-free attack
- Memory leak
- Memory management

BACKGROUND

When the memory for an object is deallocated, pointers to that object become invalid. Such pointers, known as dangling pointers, can be exploited by attackers to cause undesirable or malicious program behavior. A pointer that points to memory that has been reused (reallocated) is known as a use-after-free pointer; these, too, are gateways to exploits.

DESCRIPTION

This disclosure describes techniques that detect the presence of dangling and use-after-free pointers in runtime and, upon detection, cause a program crash, which may be preferable to leaving open the potential for security breaches. The program crash is accompanied by a bug report, enabling the debugging of memory errors, typically difficult to debug.

Per the techniques, a new pointer type, CheckedPtr<T>, is defined as follows.

```
class CheckedPtr<T> {
  const word_t generation;
  const CheckedPointee<T>* ptr;
};
```

Here, CheckedPointee<T> is the object being pointed to, and generation is a piece of metadata, further described below, that indicates the re-use status and (in)validity of the allocated memory. CheckedPointee<T>, the object being pointed to, is also augmented with the same metadata as follows.

```
class CheckedPointee<T> {
  const word_t generation;
  const T value;
};
```

```
class PointeeObject : public Mixin { ... };

class Mixin { ... };

class Foo {
  PointeeObject* pointee_;
  Mixin* mixin_;
};
```

(a)

```
class PointeeObject : public CheckedPointee,
public Mixin { ... };

class Mixin : public CheckedPointee { ... };

class Foo {
  CheckedPtr<PointeeObject> pointee_;
  CheckedPtr<Mixin> mixin_;
};
```

(b)

Fig. 1: Enabling CheckedPtrs. (a) code with ordinary pointers (b) code with CheckedPtrs

As illustrated in Fig. 1, to enable validity-checking and/or reused-memory checking, a programmer uses `CheckedPtrs` instead of ordinary pointers. There is no other change in the implementation language. Changes to code can be made incrementally, e.g., some pointer variables in a program can be of type `CheckedPtrs` while others remain normal pointers.

When memory is allocated to an object of class `CheckedPointee`, its `generation` field is initialized to a unique valid value. When a `CheckedPtr` is made to point to that object, its `generation` field is set equal to the `generation` field of the object. When the memory is deallocated, the `generation` field of the object is set to an invalid value.

If the memory is allocated, then deallocated, then again reallocated, the `generation` field of the object upon reallocation is to set it to a valid value different from the valid value that was given upon allocation. One way to ensure that the `generation` value is unique to each reallocation is to monotonically increase `generation` value with each reallocation. (Hence `generation` is also referred to as sequence number.) However, to make it harder for attackers, `generation` can also be assigned a new random value upon each reallocation.

When a `CheckedPtr` is dereferenced, e.g., its contents accessed, its `generation` field is compared to the `generation` field of the object it points to (`CheckedPointee`). If there is a mismatch in the two `generation` fields, or if the `generation` field of `CheckedPointee` is invalid, a bug report is generated and a runtime error is thrown. Comparing two `generation` fields can amount to such computationally inexpensive operations as bit-rotations, bit-masking, and bitwise AND operations.

The `generation` field enables distinguishing between valid accesses, reused memory, and deallocated memory as follows.

Valid access

A pointer dereferencing, e.g., an access of the pointer contents, is valid if its generation field matches the generation field of the object it points to:

```
CheckedPtr.generation == CheckedPointee.generation
```

Reused memory

A pointer dereferencing, e.g., an access of the pointer contents, can detect re-used memory and can throw an error if its generation field has a mismatch with the generation field of the object it points to:

```
CheckedPtr.generation != CheckedPointee.generation
```

Invalid access (deallocated memory)

A pointer dereferencing, e.g., an access of the pointer contents, is invalid if the `generation` field of the object it points to is invalid:

```
!IsValid(CheckedPointee.generation)
```

The contents of deallocated memory depend on the underlying memory allocator. For example, in some memory allocators, the first word of a deallocated slot includes an encoded pointer to the next free-list element, which has a known structure (in particular, the high-order bits have a known value, per the architecture and operating system requirements). This can be used to ensure that deallocated memory does not include a `generation` value that is valid.

Fig. 2: Using the upper 16 bits of a 64-bit architecture to store the `generation` field of a `CheckedPtr` with zero overhead

In a typical 64-it architecture, the upper 16 bits (bits 64-48) are typically set to zero, since the widest memory space is only 48 bits wide (corresponding to an addressable space of 2^{48}=280 TB). As illustrated in Fig. 2, these upper 16 bits can be used to encode the `generation` field of `CheckedPtr`, thereby enabling a 16-bit `generation` field with zero memory overhead for the pointer. Additionally, due to the nature of the `C/C++ malloc()` library function, which aligns memory allocated by it, the lower 3 bits are zero as well. This enables up to a 21-bit `generation` field without any size overhead on the pointer side. As illustrated in Fig. 3 below, `CheckedPointee`, the pointed-to object, continues to have a one-word overhead.

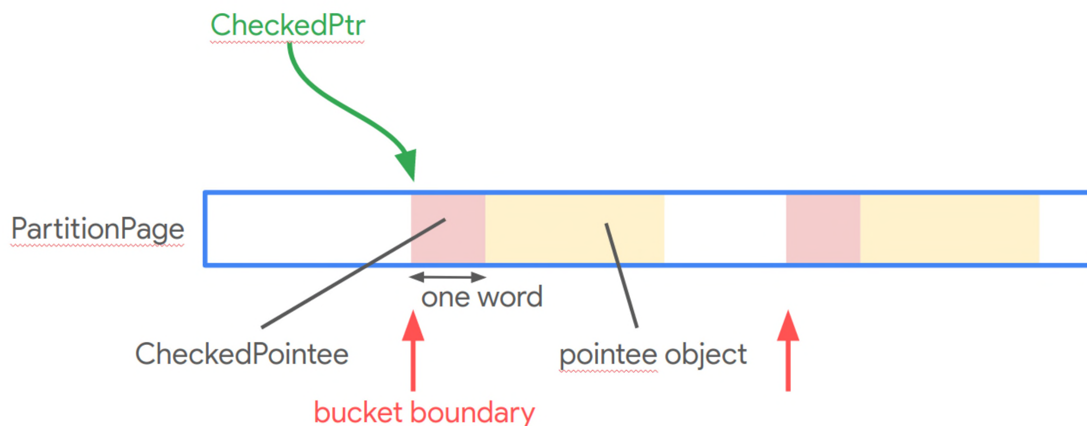

Fig. 3: Partitioning a page to include `CheckedPointees`

Fig. 3 illustrates the partitioning of a page to hold objects of type `CheckedPointee`. `CheckedPtr` (in green) is made to point to the one-word `generation` field (pink), which can be placed adjacent to the pointed-to object (yellow).

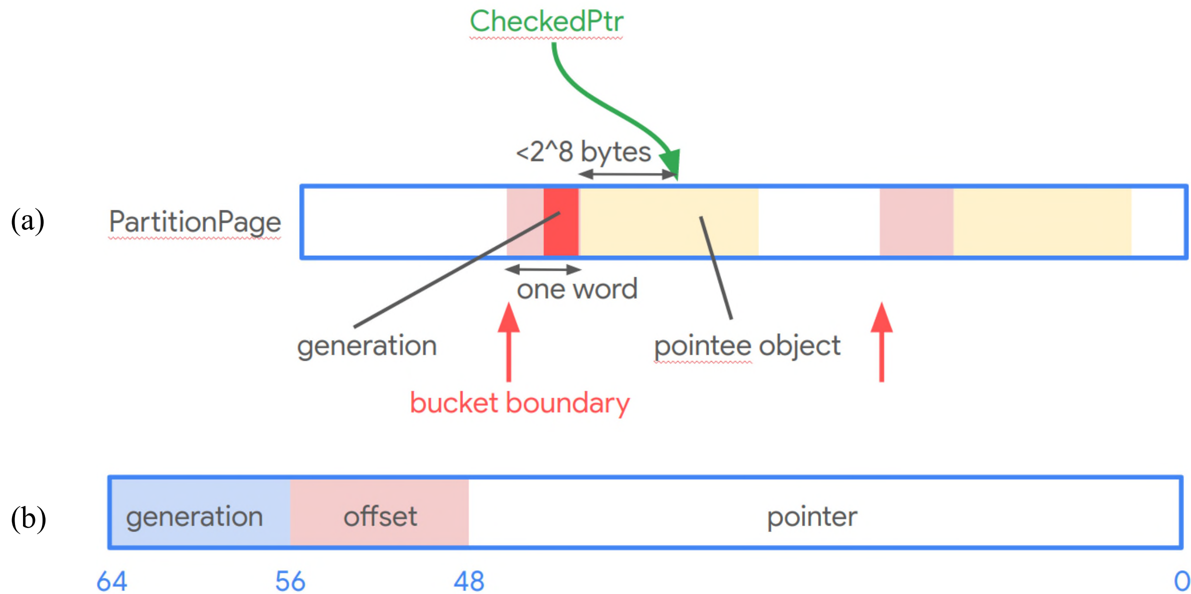

Fig. 4: A `CheckedPtr` that can point to the interior of an object (a) page partition (b) structure of the `CheckedPtr`

Fig. 4 illustrates a variant of `CheckedPtr` that can point to the interior (e.g., not necessarily to the start) of an object. Fig. 4(a) illustrates a page partition that holds the pointed-to object. Fig. 4(b) illustrates the structure of `CheckedPtr`: bits 64-48 are divided into the previously-described `generation` field and an `offset` field that indicates the point within the pointed-to object that `CheckedPtr` points to. The described variant of `CheckedPtr` can point to the middle of complex data structures such as vectors, linked lists, etc.

In this manner, the techniques of this disclosure forestall security breaches due to dangling or use-after-free pointers. The techniques require no change in the implementation language, require no atomic instructions, cause no pointer chasing, can be added incrementally, and have minimal memory (one word for the pointed-at object) and runtime overhead. The described techniques can be implemented in software such as browsers, operating systems, etc.

CONCLUSION

This disclosure describes techniques to detect the presence of dangling and use-after-free pointers in runtime and, upon detection, forestall security breaches due to such pointers by causing a program crash accompanied by a bug report. Per the techniques, both pointer and pointed-to object are augmented with metadata that enables, upon dereferencing, the checking of the validity of the pointer and the re-use status of the memory it points to.