2011 31st International Conference on Distributed Computing Systems

A New Class of Buffer Overflow Attacks

Ashish Kundu, Elisa Bertino
Department of Computer Science & CERIAS, Purdue University
{ashishk, bertino}@cs.purdue.edu

Abstract

In this paper, we focus on a class of buffer overflow vulnerabilities that occur due to the "placement new" expression in C++. "Placement new" facilitates placement of an object/array at a specific memory location. When appropriate bounds checking is not in place, object overflows may occur. Such overflows can lead to stack as well as heap/data/bss overflows, which can be exploited by attackers in order to carry out the entire range of attacks associated with buffer overflow. Unfortunately, buffer overflows due to "placement new" have neither been studied in the literature nor been incorporated in any tool designed to detect and/or address buffer overflows. In this paper, we show how the "placement new" expression in C++ can be used to carry out buffer overflow attacks – on the stack as well as heap/data/bss. We show that overflowing objects and arrays can also be used to carry out virtual table pointer subterfuge, as well as function and variable pointer subterfuge. Moreover, we show how "placement new" can be used to leak sensitive information, and how denial of service attacks can be carried out via memory leakage.

1. Introduction

Buffer overflows occur when `n` bytes are written into a memory area (buffer) of size less than `n` bytes. If an attacker gains direct or indirect control of what is written into this memory area, she can carry out buffer overflow attack(s). Buffer overflows have been one of the most widely exploited vulnerabilities, and have led to several high-profile successful attacks. Some of the (in)famous buffer overflow exploits are by the Morris Worm [23], [21], Blaster [6], Apache/mod_ssl [5] and Code Red [4]. Over 60% CERT advisories concern buffer overflow vulnerabilities[1].

Buffer overflows are primarily due to "unchecked buffers" – lack of robust bounds checking when copying strings and arrays. Even if bounds checking is in place, it may not be robust due to integer overflows and underflows. For example, let the number of characters to be copied from `char* src` to `char* dest` be `n`: `if (n > 0) strncpy(dest, src, n);`. If `n` is an `unsigned int` and is decremented in a program path reaching this use of `n`, then `n` might contain a very large value. Buffer overflows can also be effected by exploiting format string vulnerabilities.

In this paper, we present a new mechanism for carrying out buffer overflow attacks, which is *independent* of existing mechanisms. The language primitive that we exploit to carry out attacks is the *placement new* expression supported by C++ [24], [25]. The primary purpose of this expression is to "place", i.e., to allocate a dynamically created object or array at a given address that refers to a memory arena that has *already* been allocated to the process. The use of "placement new" has several advantages in terms of software engineering – (1) the program can make use of memory pools and is more efficient as it does not have to allocate memory dynamically; (2) memory allocation exceptions can be avoided, which is an important requirement in safety-critical programs; and (3) custom garbage collectors and debuggers can be built.

"Placement new" is thus a powerful expression and supports important functionalities. However, from a security standpoint, "placement new", if not appropriately used in programs may lead to security threats. When using this construct, a programmer must ensure that: (1) the size of new object being placed is not larger than the memory already allocated; (2) the alignment issues are appropriately handled; and (3) de-allocation of the memory is carried out appropriately. Such expectations are complex in nature. By allowing to "place" any object at any memory location (allocated to the process), C++ not only makes its type system weak (which is known), but also makes programming more complex.

Objects are increasingly being used as units of communication between programs. Web services, cloud-based services, and object-based interactions between browsers and servers such as Ajax/Json models employ object-based information transfer paradigms. The fact that objects are passed to a program from tainted/untrusted sources (remote/local) makes object-based buffer overflows an important concern. Programs may also make "logic errors" on placing an object in a buffer that has a size smaller than the size of the object.

Like traditional buffer overflow attacks, "placement new"-based buffer overflow attacks also occur due to insufficient and/or inaccurate bounds checking. However, such attacks are different from traditional ones, because: (1) the functionality of "placement new" is unique yet powerful; and (2) "placement new" can lead to both object and array overflows. None of the existing tools can detect buffer overflow vulnerabilities due to "placement

• *Ashish Kundu is currently with the IBM T J Watson Research Center, Hawthorne, New York, USA: akundu@us.ibm.com.*

1. http://www.mcafee.com/us/local_content/white_papers/wp_ricochetbriefbuffer.pdf

1063-6927/11 $26.00 © 2011 IEEE
DOI 10.1109/ICDCS.2011.63

new" (some of the tools are Coverity, Fortify, FlexeLint, Discover, Klocwork, PREfast, ITS4, PolySpace, Prexis, DMS, Flawfinder, CodeAssure, CodeSurfer, and Orion). Later, we show that StackGuard implementation by GCC also does not detect some such vulnerabilities.

Contributions: We show how buffer overflow attacks can be carried out by overflowing objects by using "placement new". Objects overflows occur because of received objects and/or construction of new objects. We show how copy constructors can be exploited for this purpose. A two-step attack methodology is shown in order to carry out array-based overflows. The types of overflow that can be carried out using "placement new" are: (1) stack overflow, (2) heap/data/bss overflow, which can be exploited for arc and code injection, modification of variables on data/bss and stack, modification of function pointers, of virtual table pointers, and of internal states of objects. We also show how information can be leaked and confidentiality/privacy breached due to the use of "placement new". We also show how DoS attacks can be carried out using "placement new". Memory leaks can also be effected. We have demonstrated each of the attacks described in this paper on a Ubuntu 10.04 system with programs compiled with *gcc* 4.4.3. We also discuss the basic measures that can be adopted for preventing "placement new" vulnerabilities.

2. Placement New in C++

```
void *operator new (size_t,void *p) throw() {return p;}
void *operator new[] (size_t,void *p) throw() {return p;}
```

The first constuct is used for allocating a single object/data structure, and the second one for arrays. "Placement new" expression makes it possible for the programmer to "place" an object or a dynamically allocated buffer/data structure at a specific memory area. The starting address of this specific memory area is passed as a `void *` to the `new` operator. An example of the use of this expression is as follows. `newtext` is dynamically allocated 10 bytes (sizeof(char) is one byte) starting at an address that is the value of `text`. The address must be a non-null one.

```
char *text = new char(10);
//Place newtext at the starting address of ``text''
char *newtext = new (text) char(10);//uses placement-new
```

2.1. Uses of Placement New

The "placement new" expression is useful in many respects. Its specific uses are (consider the following code snippet: `T *t = new (&buf) T();`):

1) initialize the memory allocated to `buf`, which might be a hardware address. It can be used as `memcpy()`.
2) re-use the memory allocated to an object `buf` for `t`.
3) build a custom-made memory pool for the application, which would act as a heap for the specific application. Mission-critical systems rely on memory pools and re-use of memory in order to avoid allocation failures. Custom memory management can be carried out using such a feature.
4) de-serialize "serialized" objects and place it at the memory arena of an object constructed previously. For example, suppose that the customized constructor `T(ifstream serobj)` constructs an instance of class T from the serialized object in an input stream `serobj`. "placement new" can be used as follows: `T t1; T *t2 = new (&t1) T(serobj)` places the received serialized object in the memory arena of `t1`.

2.2. Objects

"placement new" facilitates the placement of objects at a memory arena that is pre-allocated for an object or an array. A common practice is to create an instance of a superclass and then set its members by "placing" an instance of a subclass. The implicit assumption here is that the size of the instance of the subclass should not be larger than that of the instance of the superclass. Moreover, the alignment of the members must also be identical. However, such assumptions need to be enforced by the programmer(s). Maintenance of legacy code over the years may require creation of new subclasses and wrappers in order to incorporate new requirements and fixes. Instances of newly added subclasses may need to be placed at the memory allocated to the instances of older superclasses using "placement new".

Other than the common use-cases of "placement new", such memory reuse can be carried out in web services, mobile object based systems, and mobile devices, where in order to minimize overhead, objects might be placed into pre-allocated memory of a superclass object. Since the state and size of such objects can be influenced by attackers, object overflow may occur. Listing 1 presents a Student class, which is used as the running example to illustrate the attacks.

```
Class Student {
  public: Student():gpa(0.0), year(0), semester(0) { }
  private: double gpa, int year, semester;
};
Class GradStudent {
  public:
    GradStudent() { }
    GradStudent(double sgpa, int yr, int sem) {
      gpa = sgpa; year = yr; semester = sem; }
  private: int ssn[3];
};
int main(int argc, char *argv[]) {
  Student s;
  //place *gs at &s.
  GradStudent *gs = new (&s) GradStudent(4.0,2009,1); }
```

Listing 1. Placement of Objects

2.3. Arrays

"Placement new" can be used to create an array as part of an already-allocated array/buffer. The code in Listing 2 shows how a string can be copied into another through "placement new".

```
char uname_buf[SIZE];
//equivalent as above: char *uname_buf = new char[SIZE];
```

```
bool checkUname(char *uname)
  //Place a uname at the ``uname_buf''; n <= SIZE
  char *buf = new (uname_buf) char[n];
  strncpy(buf, uname, n); [...]
}
```

Listing 2. Placement of Arrays

2.4. Combination of Objects and Arrays/Strings

Memory allocated to an object can be re-used for placing an array/buffer, and vice versa (see Listing 3).

```
//equivalent as above: char *uname_buf = new char[16];
bool checkUname(char *uname) {
  //Place a string object in the memory of uname_buf[]
  string *str = new (uname_buf) string(); }
```

Listing 3. Placement of a String object in an array

2.5. Security Issues with Placement New

Important issues with "placement new" are:

1) "placement new" allows any address allocated to the process to be used to place an object. The following code snippet illustrates the point: `char c; int *b = new (&c) int;`
2) "placement new" does not enforce any bounds checking. Neither compile-time or runtime enforcement of bounds checking is applied.
3) Invocation of placement new does not carry out any type-checking. If memory is allocated to an instance of type T_1, then placing an instance of type T_2 at that memory succeeds even if T_2 is not a compatible type of T_1.
4) "placement new" is used to populate an object or a data structure from a serialized instance (received from another source, possibly). However, since it does not enforce any checking of alignment, it may lead to incorrect semantics, and to program termination.
5) "placement new" may lead to memory leaks.

3. Object Overflow

The fault based on "placement new" allows an attacker to place a larger object in the memory allocated to a smaller object. Placing a larger object in the same memory arena allocated for a smaller object leads to "object overflow", that then overwrites certain memory locations.

There are two ways to effect such a fault: (1) when the program constructs a larger object in the place of a smaller object, but does not carry out bounds checking on sizes, or (2) when the program accepts an object from another program (local or remote) or from network, and places it in the place of another program. In both cases, the smaller object is generally an instance of a supertype and the larger object is an instance of a subtype. A programmer may ignore checking the sizes before placement, as he might not think it is needed. Consider the following scenario: the "extra" members defined by the subclass would not be used and therefore would not be set by anyone, which why even if the size of new object maybe larger, it is harmless. If there is a way for the attacker or a malicious program to set some or all of the extra members to their chosen values, they can overflow the object in a "meaningful" manner. In case of (2), if the object is being created by an untrusted program such as a third party web service or a Javascript/AJAX/JSON[2], it can effect an attack on the receiving program. The programmer may not include any code to check the size because of the trust on the protocol based on which objects are received and sent from one program to another.

3.1. Overflow Via Construction

An object obj of a class B is constructed using placement new; the runtime class from which the object is constructed from could be a subclass B of another class A via single or multiple inheritance. `address_of_any_variable` can be the address/reference of/to any scalar variable, of any object, or any type of array, or of type void *, but cannot be NULL.

```
B *obj = new (address_of_any_variable) new B();
```

In the Listing 4, `stud` is an object of class `Student`. The program, at certain point, creates an instance `*st` of `GradStudent` at the place of `stud`; however it does not check the size of `*st` against the size of `stud`.

```
Student stud;
GradStudent *st = new (&stud) GradStudent(gpa,...);
```

Listing 4. Object Overflow Via Construction

3.2. Overflow Via Serialized/Remote Object

When the input from the user or a file/device/network is used to set the value of certain members of the larger object, attacker-chosen values can be used to overwrite variables, other objects, function pointers, as well as virtual table pointers. Mobile applications and software that have to work within constraints on processing power and memory would rely on "placement new". Web browsers/clients send objects via java scripts/Ajax applications; one such object model is JSON. Web applications developed with less care can send a JSON object of a larger size than what is normally expected by a server. Attacks such as DOM injection can be carried out in order to overflow objects at the server side. Symmetrically a malicious webserver or website can carry out such attacks on the client computers.

Whenever a remote object influences (directly/ indirectly) the size of the object being constructed via "placement new", an object overflow attack can be carried out.

There are various mechanisms for influencing remote objects: (1) by aliasing of a variable to provide a different size of the memory supplied to "placement new"; (2) by returning an array or list whose size is not known statically nor enforced; (3) by passing a larger object.

2. http://www.json.org

A simple way in which a remote/serialized object may influence the construction of a new object is by influencing the number of items being created/populated. Listing 5 shows an example in which the program gets an array of objects/entries such as `names[]` (string type) from a third-party (malicious) service `service`, and creates an array of objects `stnames` from `names[]`. Listing 6 shows the code to generate an overflow by copying all the fields of a `Student` object to a memory pool.

```
//service.getNames() returns tainted list
string[] names = service.getNames();
//n: length of received names[]: maliciously changed
//*st is a memory pool used by this program
string[] stnames = new (st) string[n]; [...]
```

Listing 5. Object Overflow Via Construction

```
void addStudent(Student *remoteobj) {
  [...]
  GradStudent *st = new (&stud) Student();
  int i=-1;
  while (++i<remoteobj->n) {
  //int courseid is declared in class Student.
    *(st->courseid + i) = *(remoteobj->courseid + i);
  }
}
```

Listing 6. Object Overflow Via Construction: Copy

C++ supports the concept of copy constructor, using which a shallow copy of the object can be created. Such a constructor can be used to copy the values from one object into the new one(s). In the code in Listing 7, a `GradStudent` object is created after checking its type, and is placed at the memory arena of `stud`.

```
void addStudent(Student *remoteobj) {
  Student *st = new (&stud) GradStudent(remoteobj);
  [...]
}
```

Listing 7. Object Overflow: Copy Constructor

However, since the default copy constructor provided by C++ for a class does a shallow copy, a programmer may defines her own copy constructor that does a deep copy of the received/serialized object. In that case, the deep-copy construction of a `Student` object `Student *copyst = new (&stud) Student(remoteobj)` may overflow certain objects (and buffers) inside the `Student` object, if size/bounds checking is not in place. Such an overflow could occur when instead of placing the new object at the starting address of `stud`, the new object is placed in a memory pool or an existing buffer (re-used by the program).

3.3. Indirect Construction

In an indirect scenario, the `remoteobj` is not directly used to create an instance using "placement new"; instead it indirectly affects the size of the instance being used. For example: (1) there is a data flow path (intra-procedural or inter-procedural) from `remoteobj` to another object `obj` at program point p such that at p, `remoteobj` is used to update/construct `obj` so that it increases the size of `obj` (see Listing 8), and (2) `obj` is used via a copy constructor to construct an instance of `Gradstudent` based on "placement new" as in the code above, and the size of obj is larger than the size of the buffer allocated at the starting address of `stud`. It also would lead to an object overflow due to a `remoteobj`.

```
void addStudent(Someclass *remoteobj) {
  Someclass *obj2 = new Someclass(remoteobj);
  [...] //obj2: reachable at the next statement
  GradStudent *st = new (&stud) GradStudent(obj2);}
```

Listing 8. Object Overflow Via Indirect Construction

Another way for such an overflow to occur via indirect construction is when copy constructor is used in the manner shown in Listing 9. An object of class `B` might be constructed as an aggregate of several other objects, variables, arrays and/or structures. If the size of any of these component entities increase beyond the what is expected for the class, then an overflow might occur.

```
A obj2 = B();//size of B > size of A.
Student stud;
//Uses Student::Student(A)
//obj2 is copied into an instance of A in Student.
Student *st = new (&stud) Student(Obj2);
```

Listing 9. Object Overflow Via Indirect Construction: Copy Construction

3.4. Types of Overflows

Object overflows can be of two types: external and internal.

External Overflow: In an external overflow, the object overflow overwrites memory locations that are external to that object. For example, in Listing 4, the overflow of `st` is an external overflow. All of the overflows we have described till now are external overflows.

Internal Overflow: In an internal overflow, the object overflow overwrites memory locations that are internal to that object. For example, in the following Listing 10, overflow of `st` is an internal overflow. The class `MobilePlayer` in this example is constructed for players, who are students.

```
class MobilePlayer {
  Student stud1, stud2; int n;
  void addStudentPlayer(Student *stptr) {
    GradStudent *st = new (&stud1) GradStudent(stptr);
    ++n; [...] }
};
```

Listing 10. Internal Overflow

Internal overflows have the capability to modify internal states of an object, as well as to carry out each of the above attacks. In what follows, the attacks that we describe do not depend on any vulnerability other than the "placement new" vulnerability as specified.

In what follows, we show how to carry out standard overflows and other attacks using only one methodology - the object overflow via construction. Note that the object overflow via serialized/remote objects can be easily used to demonstrate such attacks as well.

3.5. Data/bss Overflow

In our example we show that an object of a subclass `GradStudent` (Listing 11) is used to populate the memory of the object of its superclass `Student`. However, note that an object of any class can be placed in the memory arena of an object of any class, or in fact starting at any address.

`GradStudent` contains another member array, `ssn[]` that stores the SSN of a graduate student. In the code Listing 11, instances `stud1` and `stud2` are allocated in data/bss area (ELF format[3]) (precisely in the bss area as they are not initialized). Statement `addStudent(false)` creates `stud2` as an instance of `Student`. Statement `addStudent(true)` creates `stud1` as an instance of `GradStudent`. In the execution of the code segment `if (isGradStudent) ...`, the values of `ssn[]` are being set using attacker-specified inputs, thus changing the value of `gpa` in `stud2`.

```
Student stud1, stud2;
bool addStudent(bool isGradStudent) {
  GradStudent *st;
  if (isGradStudent) {
    //user input:ssn[0],ssn[1],ssn[2]; place st at &stud1
    st = new (&stud1) GradStudent(gpa,...);
    st->setSSN(...); }
  else {
    //user input:gpa, year, semester; place st at &stud2
    stud2 = new (&stud2) Student(gpa,...); }
}
[...]
addStudent(false);
addStudent(true); //attack: overwrites ``gpa'' of stud2.
```

Listing 11. Data/bss Overflow

3.5.1. Heap Overflow. Heap overflow is similar to the data/bss overflow. Listing 12 shows an example of a heap overflow attack. The value of `name` is overwritten by `ssn[]` of `stud` on the heap. It can further make the program more vulnerable to attacks that can be carried out using heap overflows.

```
Student *stud; char *name;
int main() {
  GradStudent *st = new (stud) GradStudent();
  name = new char[16];
  strncpy(name, abcdefghijklmno\0'',16);
  cout<<``Before Attack: Name:``<<setw(16)<<name<<endl;
  cin >> st->ssn[0];cin>>st->ssn[1];cin>>st->ssn[2];
  cout<<``After Attack: Name:``<<setw(16)<<name<<endl;
  return 0; }
```

Listing 12. bss Overflow

3.6. Stack Overflow

In stack overflow, the idea is that the memory arena of a local object is used to place a larger object, and the program does not have any checking in place for the size of the object.

3.6.1. Modification of Return Address. In the code in Listing 13, a larger size object `gs` is being placed in the memory arena of `stud`. By doing so, in the call stack, the return address of `addStudent()` is being overwritten by `ssn[0]` (If the frame pointer is saved, then `ssn[1]` would overwrite the return address.) If the system employs canaries (such as the StackGuard in gcc) in order to protect from stack overflow, then `ssn[2]` overwrites the return address. Here, we assume that the size of each of the addresses (frame pointer) and the canary is same as the size of an int (4 bytes in Ubuntu Linux).

```
void addStudent(bool isGradStudent) {
  Student stud;
  if (isGradStudent) {
    GradStudent *gs = new (&stud) GradStudent();
    int i=-1, dssn=0;
    while (++i < 3) {
      cin >> dssn; if (dssn>0) gs->ssn[i]=dssn; }}
}
[...]
addStudent(true);
```

Listing 13. Modification of Return Address

Leaving the canaries unmodified: An attacker can exploit the above code segment to overwrite *only* the return address and not the canary[4]. This can be achieved in this case by supplying non-positive values for first two iterations of the while loop. The third one would be supplied with the new return address.

3.6.2. Arc and Code Injection. Since `ssn[]` overwrites the return address of `addStudent()`, the attacker can carry out an *arc injection* attack (same as return-to-libc attacks) [22] by specifying the address of another method in the same code. For example, the address of a method that makes a system call in a privileged mode can be used. If the size of an instance of `GradStudent` is large enough to overwrite the return address, and the size of all local variables in `addStudent()` is enough to inject shell code, then the attacker can set the values of `ssn[]` and other variables (e.g., `stud`) so that the function would return to execute the supplied shell code. If the process is running in privileged mode, the attack would have a significant impact on the integrity of the system and confidentiality of data.

3.7. Modification of Variables

Overwriting variables that are allocated after `stud1` can be carried out in the same manner as described below.

3.7.1. Variables in data/bss. In the code in Listing 14, the value of `noOfStudents` is overwritten by `ssn[]` of `stud1`. It further can make the program more vulnerable to other form of attacks. We discuss such an attack in Section 4.

```
Student stud1; int noOfStudents = 0;
bool addStudent(bool isGradStudent) {
  GradStudent *st;
  if (isGradStudent) {
    //input:ssn[0],ssn[1],ssn[2]; Place st at &stud1
    st=new (&stud1) GradStudent(gpa,...);st->setSSN(...);}
  else {
    //input: gpa, year, semester; place st at &stud2
```

3. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

4. gcc v3.4 includes implements StackGuard [8] that uses canaries.

```
    st = new (&stud2) Student(gpa,...); } [...]
}[...]
addStudent(true); //attack: overwrites ``noOfStudents''.
```

Listing 14. Modification of Data/BSS variables

3.7.2. Local Variables on Stack. Local variables declared before `stud` in a function can be overwritten by overflowing the object `stud`. Consider the code in 15. A call to `addStudent(true)` pushes `n` and then `stud` to the call stack. `ssn[1]` overwrites `n`.

```
void addStudent(bool isGradStudent) {
    int n = 5; Student stud;
    if (isGradStudent) {
          GradStudent *gs = new (&stud) GradStudent();
      [...] }
    for (int i=0; i< n; i++) { [...] }
}[...]
addStudent(true);
```

Listing 15. Overwriting Local Variables on Stack

Alignment Issues: It is necessary to note that the memory for `n` is allocated with a 4-byte alignment. `ssn[0]` does not overwrite n, but `ssn[1]` overwrites `n` because `stud` as an instance of `Student` does not end exactly at the 4-byte alignment; it leaves 4 bytes for padding, which is occupied by `ssn[0]` as part of `*gs`, which is an instance of `GradStudent`.

3.8. Modification of Objects

Objects can be overwritten via object overflow. By overwriting objects partially or fully, member variables as well as virtual table pointers can be overwritten. Modification of objects can be carried out via stack, heap or data/bss overflows.

3.8.1. Modification of Member Variables. With reference to the `addStudent()` method in Listing 15, suppose that an object is declared in the place of `n`:
`Student first=Student(3.9,2008,2).first.gpa` can be overwritten by placing object `*gs` instance of `GradStudent` at the memory arena of `stud`. In such a scenario, `first.gpa` is overwritten by `gs->ssn[0]`. Listing 16 presents the code snippet for this attack.

```
void addStudent(bool isGradStudent) {
    Student first = Student(3.9, 2008,2);
    Student stud;
    if (isGradStudent) {
      GradStudent *gs = new (&stud) GradStudent();
      cin >> gs->ssn[0]; //overwrites first.gpa
      cin >> gs->ssn[1];} [...]
}[...]
addStudent(true);
```

Listing 16. Overwriting Member Variables of Objects

3.8.2. Virtual Table Pointer Subterfuge. Virtual tables (vtables) are part of each instance of each class and are used to carry out dynamic dispatching of invocation of virtual functions. The compiler creates a vtable and adds a pointer to this table in each instance of each class. The virtual table maps a virtual function to the address of the actual implementation of the function that should be invoked.

Suppose that a virtual function `virtual char* getInfo()` is added to both classes `Student` and `GradStudent`. The virtual table would contain one row: first entry is `virtual char* getInfo()`, which points to the specific implementation. The virtual table of `GradStudent` contains the address of the local implementation
`char* GradStudent::getInfo()`, while that of the base class `Student` contains the address of the local implementation of `char* Student::getInfo()`.

The C++ compiler adds a pointer to the virtual table `*__vptr` in each instance as the *first entry*. The memory location at the 0'th offset inside an instance of `Student` or `GradStudent` contains `*__vptr`. Therefore, any overflow that can overwrite an object can in fact overwrite the virtual table pointer. In case of multiple inheritance, there are more than one vtable pointers in a given instance.

Via Data/bss Overflow: In our example in Listing 12, `stud1` overflows into `stud2`. In the updated definitions of classes with the virtual function, the first entry in the object `stud2` is not `gpa`, but `*__vptr`. The overflow of `stud1` would overwrite the `*__vptr` with the user-supplied value of `ssn[]`. Such an overflow allows the attacker to invoke arbitrary methods as implementations of
`virtual char* getInfo()` or even crash the program by supplying an invalid address as the value of `*__vptr` or of the virtual method.

Via Stack Overflow: As in the case of data/bss overflow, in Listing 16, the virtual pointer of first (`first.__vptr`) would be overwritten.

3.9. Function Pointer Subterfuge

Overwriting function pointers is an important goal of attackers. A function pointer that is defined in data/bss or stack can be modified in the same was as variables and objects are modified. Consider the code in Listing 17:

```
void addStudent(bool isGradStudent) {
  bool (*createStudentAccount)(char *uid) = NULL;
  Student stud;
  if (createStudentAccount != NULL)
    createStudentAccount(...); [...]
} [...]
addStudent(true);
```

Listing 17. Function Pointer Subterfuge

By overflowing object `stud`, an attacker can make the function pointer `*createStudentAccount()` to point to an arbitrary location, and cause the invocation of the code in that location. Note that the function would not be invoked if it were assigned a null value. Therefore, such an attack also enables invocation of a method that was not supposed to be called in a given context.

3.10. Variable Pointer Subterfuge

Overwriting variable pointers is another important goal of attackers. The example for function pointer subterfuge can be slightly modified to carry out this attack; the modified code is presented in Listing 18.

```
Student stud; char *name;
int main() {
  GradStudent *st; name = new char[16];
  st = new (&stud) GradStudent();
  cin >> st->ssn[0]; //overwrites ptr name
  cin >> st->ssn[1]; cin >> st->ssn[2]; return 0;}
```

Listing 18. Variable Pointer Subterfuge

By overflowing the object `stud`, an attacker overwrites the value of `name`, which essentially is the starting address of a 16-char array. The pointer subterfuge makes the variable point to an arbitrary location, and causes the program to crash or use an attacker specified value at another location.

4. Array Overflow

A local variable might also be used to allocate memory, and can be used to carry out a stack overflow. Overflow of array-type buffers such as strings is commonly exploited by the attackers. Using "placement new", an attacker can mount such attacks in two steps.

The idea is that a memory pool is already created and any new buffer needed is created out of that memory pool using "placement new". The constraint is that the size of the buffer is never greater than the size of the memory pool. In the first step of the attack, the attacker modifies the variable that stores the size of the buffer to a value larger than the memory pool size by overflowing an object using the appropriate method(s) specified above. The attacker would then be able to modify the data/bss/heap or stack locations that are beyond the memory pool. That way, code or arcs can be injected, return-to-libc-attacks can be easily carried out. In the next step, the user passes in a maliciously crafted string to the buffer as it is done in case of traditional buffer overflow scenarios - such as to open a remote shell. One can think such a vulnerability not to be so much dangerous as its exploitation requires two steps. However, we should keep in mind that for attackers, in order to take control of a system, a network or some sensitive data, making two steps is not a hard task.

Now consider the following example. A function `bool sortUname(char *uname)` takes as input a list of user names separated by a newline characte, sorts them, and then adds them to the database, if they are valid. The number of user names `n_unames` is always less than the number of students `n_students`.

4.1. Stack Overflow using Arrays

```
bool sortAndAddUname(char *uname)
  char mem_pool[n_students*(UNAME_SIZE+1)]; //+1 for '\n'
  int n_unames=0; Student stud; cin >> n_unames;
  if (n_unames > n_students) return; [...]
  if (isGrad) {
    GradStudent *st = new (&stud) GradStudent();
    //read st->ssn[] from std input or from file;
    //use it to validate a grad student.
  [...]}
  char *buf = new (mem_pool) char[n_unames*(UNAME_SIZE+1)];
  strncpy(buf, uname, n_unames*(UNAME_SIZE+1)); [...]}
```

Listing 19. Stack Overflow involving Arrays

The first step is carried out by the `if isGrad` block. `n_unames` can be set to an arbitrary value by the input values for member variable `ssn[]` in the instance. Now when it is set to a value larger than `n_students`, the size of `buf` is larger than the size of the `mem_pool` array. The use of `strncpy` is perfectly secure when we ignore the object overflow scenario. However, after the overflow, more characters are copied to the memory arena of `mem_pool[]` than it can hold, thus leading to overwriting the return address. The attacker can now gain access to the system.

4.2. Heap/data/bss Overflow using Arrays

Heap, data, and bss overflows can be also carried out in this manner. For heap overflows, `mem_pool` is dynamically allocated; for data/bss, `mem_pool` are arrays declared globally and not part of any function or method. The following code overflows `mem_pool` after `n_unames` is overwritten by object overflow.

```
char mem_pool[n_students*(UNAME_SIZE+1)]; int n_staff;
bool sortAndAddUname(char *uname)
  int n_unames=0; Student stud;
  //remaining code segment is same as in previous example.
  [...] }
```

Listing 20. BSS Overflow involving Arrays

4.3. Information Leakage

In this section, we show how information can be leaked via "placement new". A given memory pool is used to hold different objects or data structures at different program points. Information leak can occur when a smaller object is allocated in the memory pool, where a larger object was allocated earlier. The "placement new" operator facilitates carrying out such operations, without however sanitizing the bits of the memory pool. If the attacker can control the size of second object/array, then information leak would occur.

Consider the code in Listing 21, in which the password file is read into `mem_pool`; later a string from the user is stored in `mem_pool`. However, since the attacker (user) may pass a string that is of size less than SIZE (of `mem_pool`), the remaining part of the memory still has the contents of the password file.

```
char mem_pool[SIZE]; char *userdata;
int main(int argc, char *argv[]) {
  //mmap/read a  password file to mem_pool.
  //MAX_USERDATA <= SIZE
  userdata = new (mem_pool) char[MAX_USERDATA];
  //user input: userdata, sizeof(userdata) <= MAX_USERDATA
  [...]
  //stores memory contents starting at userdata.
  store(userdata); }
```

Listing 21. Information leakage Via Arrays

Similarly, objects can lead to information leakage (see Listing 22). An instance `gst` of class `GradStudent` is created; later when `gst` is no longer needed, the memory arena of `gst` is re-used for creating an instance of class `Student`, which is of lesser size than the size of `gst`. Therefore `ssn[]` of `gst` remain in the memory, as "placement new" does not clean the data from the memory.

```
GradStudent *gst;
int main(int argc, char *argv[]) {
  gst = new GradStudent(); //contains SSN
  [...]
  Student *st = new (gst) Student(); //does not clean SSN
  store(st); //stores memory contents starting at st.
}
```

Listing 22. Information Leakage Via Objects

4.4. DoS Through Overflow

In this section, we show how denial of service attacks can be carried out by exploiting "placement new" vulnerabilities. It is well-known [22] that by overwriting the local variable in the data/bss or on the stack and changing its value appropriately, a specific condition that is part of the control flow can be modified to hold true or false at the attacker's discretion. For example, in Listing 15, the local variable `n` is defines the iteration count for the `for` loop. In the previous section, we have shown how to overwrite `n` using `ssn[]`. By modifying `n` to a non-positive value, or a very large positive value, the loop can be controlled such that either it is never taken or is iterated for a long time, respectively. If the loop is at the server side, it would significantly affect the response time, and/or might not allow other requests to be served. Moreover, if the resources are allocated/locked inside the loop, the attacker may crash the program, may effect memory leakage, deadlocks (trying to lock the same resource multiple times), or might crash the whole software stack on which it is running on (by using up all the memory, or opening maximum number of files or creating maximum number of processes). Authentication mechanisms can also be bypassed as is carries out via buffer overflow.

4.5. Memory Leaks

If memory allocated using "placement new" is not properly managed, there could be memory leaks. An attacker may exploit certain conditions of the system in order to hasten the process of such leakage thus crashing the system. In the Listing 23, the amount of memory released from `st` is of the size of an instance of `Student`, while the amount of memory allocated was for an instance of `GradStudent`. The amount of memory leaked per iteration is the difference in the size. Memory management is made harder by the fact that C++ does not support a "placement delete" while it supports "placement new". Even though it is widely recommended that whenever a program uses "placement new", the program should define its own "placement delete", this recommendation is rarely followed.

```
GradStudent *stud = NULL;
void addStudent() {
  for (int i=0; i<n_students; i+=2) {
    *stud = new GradStudent();[...]
    Student st = new (stud) Student();
    stud = null; [...] //free memory of st.
  }}
```

Listing 23. Memory Leaks

5. Protection Techniques

We now discuss how to protect from "placement new" vulnerabilities. We first address the case of new software under development, and thus not yet deployed, and of software which is deployed but for which it is possible to modify the source and generate a new image for deployment. We refer to such case as modifiable software. We then discuss the case of legacy software, that is, the case of software for which the source code is not available and of software that even if modifiable cannot be re-deployed.

5.1. Modifiable Software & Correct Coding

There are several approaches for static detection and prevention of buffer overflows in programs [10], [15], [26]. Vulnerabilities related to "placement new" are not part of the buffer overflow taxonomy and thus have not been by such approaches. Also, in order to carry out bounds checking for protection against "placement new" vulnerabilities, static detection schemes should be able to infer the buffer size even in cases when it is not explicit. "Placement new" allows one to place objects and arrays at a location that may just be a scalar variable and may not be a lexically defined array. The problem gets more complex with the issues of aliases – a pointer could have been assigned the address of a scalar variable or an array at any given point of program execution. Therefore static analysis of programs may not always succeed in precisely determining the size of the buffer. Therefore, the most suitable approach to address "placement new" vulnerabilities is to use correct coding [9]. In what follows, we describe the elements of "correct coding" for addressing "placement new" vulnerabilities.

Object Overflows: The primary mode of overflow via "placement new" is object overflow. In order to prevent of object overflows, the sizes of instances have to be checked. At each point, where "placement new" is used, it has to be enforced that the size of the new object or array `B` being placed in a memory arena of another object/array `A` should never be larger than the object or array A. If the size checking fails, then the memory allocated to `A` should be freed, and the non-placement "new" expression should be used to create `B`. In order to determine the size of objects, `sizeof()` operation should be used; manual estimation of size of an object should be avoided. Compilers often add member variables such as the virtual table pointer to a class, which influences the size of objects.

Memory Leaks: In order to prevent memory leaks due to "placement new", there are several options. One option is

that the programmer defines an appropriate placement delete and invoke it whenever it has to release the memory allocated by "placement new". The other option for the programmer is to ensure that the size of the objects being constructed in a given memory arena is identical to that of the memory arena. However, this is not quite a practical approach. The programmer in such a case, should make the first pointer to the memory arena equal to the null pointer only after the memory arena is not any longer used, and the pointer has been freed (by using an appropriate de-allocation method). Note that this memory arena could have been allocated using `malloc()`, and "placement new" can be used to allocated chunks of this arena to objects/arrays. We find this last option to be the easiest one to implement, while the first one requiring the implementation of "placement delete" to be a complicated for most programmers.

Information Leaks: In order to prevent information leaks, memory needs to be sanitized. Before a memory arena allocated to pointer `A` is allocated to another pointer `B`, `memset()` or its other variants should be used to set the memory to uniform bit patterns; the most common values used for memory sanitization are either 0 or 1. For efficiency sake, the programmer might be tempted to sanitize not the whole memory but only the chunk of memory not to be occupied by `B`. However, determining this may not be straightforward. Consider the following scenario: memory of `A` already contains the student id's and their SSN's. `B` would use only the alternate word (e.g., word = 4-bytes) memory (in case of an array, odd-numbered indices). In that case, the programmer should ensure that other memory words are sanitized appropriately. This would get complicated, when memory alignments are taken into account. Memory alignments ensure that integers start always at the 4-byte boundaries, and so on. Sometimes there are minor variations between compilers on how alignment is carried out. Suppose that `A` is a pointer to an instance that represents student id's and SSN's as characters. However, suppose that `B` contains a mix of integers and characters. In such a case, there would be bytes that would be used as paddings in order to properly align the member variables of `B`. The bytes used for padding might contain data from `A`.

5.2. Legacy Software

Even though a programmer can define overloaded versions of "placement new" with more sophisticated capabilities, in this paper we have focused only on the semantics and syntax offered by the "placement new" as part of the standard C++.

StackGuard employed by gcc protects return addresses on stacks for "placement new" vulnerabilities. During our experiments, we found that our attempts at stack-smashing were detected by the code that was compiled by gcc, and the program was terminated. We then carried out experiments to see whether we could selectively overwrite the return addresses, and avoid modification of the canary. We succeeded, and StackGuard could not detect it ([8] already mention that such a case may occur). Nevertheless, this experiment corroborates the fact that protecting return addresses dynamically can be successful. Therefore binary instrumentation of legacy software [19] can be used in this case also.

In order to provide non-executable stacks, a possible approach is to use a return address stack, which holds the return addresses of functions [27], [20]. Schemes that carry out runtime prevention of buffer overflow are the most suitable for legacy code. Library-based protection approaches such as using "libsafe" and "libverify" would not require recompilation of software and are suitable for legacy code, and can be updated appropriately to intercept dynamic invocations to "placement new" and carry out bounds checking. However, as we mentioned earlier, bounds checking may not be as easy here because "placement new" just operates on an address, not on a lexically declared array.

6. Related Work

There are several papers that propose a texonomy of vulnerabilities [1], [2], [3], [12], [14], [16]. A recent paper by Bishop et al. [3] defined a set of preconditions for buffer overflow attacks. They define two sets of preconditions for four types of attacks: (1) executable Buffer Overflows, and (2) data buffer overflows. An executable buffer overflow occurs when an attacker is able to place some instructions in memory and get them executed in the control flow of the process. The overflow schemes using "placement new" that we have presented in this paper supports such preconditions. Buffer overflows are carried out primarily by overflowing arrays/string buffers, when bounds checking are not in place [9], [11]. Integer overflows and underflows [22], and format string vulnerabilities are also exploited to overflow buffers [17]. Existing works on buffer overflow vulnerabilities in C++ [28], [26], [9], [22] have focused on these vulnerabilities.

Aleph One's article [18] explained lucidly on how to carry out buffer overflow attacks in C and C++. Later, Klog [13] showed that even an overflow by one byte on the stack can be used to modify the frame pointer and carry out code injection. Conover [7] showed how heap overflows can be carried out. Buffer overflow based worms have been studied by Spafford [21][23] following the Morris worm. For a detailed treatment of buffer overflows, we refer the reader to [3], [9], [22]. However, to the best of our knowledge, no existing work has explored the buffer overflow vulnerabilities resulting from the use of the "placement new" expression in C++.

7. Conclusions and Future Work

The paper shows how "placement new" expression in C++ can be a source of software vulnerabilities leading to buffer overflow attacks. To the best of our knowledge, this is the first work that studies the vulnerabilities and related attacks due to "placement new". Existing tools do not detect nor address "placement new"-based buffer overflows. A memory arena allocated to an object may not be of enough size to hold an object that is newly constructed or that is received and placed at this memory arena using "placement new". Moreover, an

attacker can supply the values of the member variables of an object such that the overflown memory locations can hold values that are meaningful to a specific attack. We have shown that an attacker can carry out stack overflows as well as data/bss/heap overflows. We have also shown that by such overflows return addresses, function pointers, virtual table pointers, and variables can be maliciously modified, and sensitive information can be leaked. DoS attacks can be carried out by memory leaks as well as by modifying variables and function pointers via object overflow. We are currently building a tool for static analysis of code and for detecting vulnerabilities due to "placement new", and automatically addressing these vulnerabilities.

References

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical report, NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

[2] T. Aslam. A taxonomy of security faults in the unix operating system. Technical report, Masters Thesis, COAST Technical Report 95-09, Computer Science, Purdue University, August 1995.

[3] M. Bishop, D. Howard, S. Engle, and S. Whalen. A taxonomy of buffer overflow preconditions. Tech Report: CSE-2010-1, Computer Science, UC Davis, January, 2010.

[4] CERT/CC. code red worm exploiting buffer overflow in iis indexing service dll. CERT Advisory CA-2001-19, July 2001.

[5] CERT/CC. Apache/mod_ssl worm. CERT Advisory CA-2002-27, September 2002.

[6] CERT/CC. W32/blaster worm. CERT Advisory CA-2003-20, August 2003.

[7] M. Conover. w00w00 on heap overflows. w00w00 Security Development (WSD), Available online at http://www.w00w00.org/files/articles/heaptut.txt., January 1999.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX SSYM*, pages 5–5, Berkeley, CA, USA, 1998.

[9] C. Cowan, P. Wagle, C. Pu, S. Beattie, , and J. Walpole. , buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition*, January 2000.

[10] D. Evans. Static detection of dynamic memory errors. *SIGPLAN Not.*, 31(5):44–53, 1996.

[11] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities. In *In Proceedings of the Network and Distributed System Security Symposium*, 2003.

[12] R. B. II and D. Hollingworth. Protection analysis: Final report. Technical report, ISI/SR-78-13, Information Sciences Institute, University of Southern California, May 1978.

[13] K. (klog@promisc.org). The frame pointer overwrite. *Phrack Magazine*, September 1999.

[14] C. E. Landwehr, A. R. Bull, J. P.McDermott, , and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211254, September 1994.

[15] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX SSYM*, pages 14–14, Berkeley, CA, USA, 2001.

[16] K.-S. Lhee and S. J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33:423–460, April 2003.

[17] T. Newsham. Format string attacks. Guardent, Incorporated. http://www.thenewsh.com/˜newsham/format-string-attacks.pdf, September 2000.

[18] A. One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), August 1996.

[19] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *USENIX Annual Technical Conference*, pages 211–224, 2003.

[20] R. G. Ragel. Architectural support for security and reliability in embedded processors. Technical report, Ph.D. Thesis, University of New South Wales, Sydney, Australia, August 2006.

[21] J. A. Rochlis and M. W. Eichin. With microscope and tweezers: the worm from mit's perspective. *Commun. ACM*, 32(6):689–698, 1989.

[22] R. Seacord. *Secure coding in c and c++*. Addison-Wesley Professional, 2005.

[23] E. H. Spafford. Crisis and aftermath. *Commun. ACM*, 32(6):678–687, 1989.

[24] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Professional, 9 edition, 1994.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3 edition, 2000.

[26] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *ACSAC*, 2000.

[27] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, 2003.

[28] Y. Younan, W. Joosen, and F. Piessens. Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Technical report, CW386, Computer Science, Katholieke Universiteit Leuven, July 2004.