# Homework 9

```
function removeLast() public {
        require(0 <= records.length);
        records.length--;
}
```

There's an off-by-one error that allows for underflowing the `records.length` variable. This causes an arithmetic underflow, effectively disabling Solidity's array bounds checking. As a result, after the overflow writes to the array can be used to overwrite any storage element located after the array , in this case, is `owner`.

1) how an attacker can exploit it
All the attackers would need to do is:
• Call `popBonusCode` to underflow (Note: Solidity lacks a built-in pop method)
• Compute the storage location of `manipulateMe`
• Modify and update `manipulateMe`'s value using `modifyBonusCode`
In practice, this array would be immediately pointed out as fishy, but buried under more complex smart contract architecture, it can arbitrarily allow malicious changes to constant variables.

2) how the contract can be fixed to patch the vulnerability
The container data structure can be used here, in this contract. For uint[] private records; We can instead create a structure,
eg.
```
struct RecordStruct {
      unit record;
      unit index;
   }
```

and change
```
uint[] private records;
```
to
```
mapping(address => RecordStruct) public recordStructs;
address[] public recordList;
```

which will provide a Mapped Structs with Delete-enabled Index, which could count the records and logically control the size of the active list with delete function.

We can add a function which checks if keys do or don't exist in one step, like this:
```
function isRecord(address recordAddress) public constant returns(bool
isIndeed) {
    if(recordList.length == 0) return false;
    return (recordList[recordStructs[recordAddress].index] ==
recordAddress);
}
```

Once we do the delete function, we can first check if the key does or doesn't exist by the function we designed at the previous step and then drop the last row of the index.
```
function removeLast() public returns(bool success) {
```

```
      address keyToRemove = recordList[recordList.length-1];
      if(!isRecord(keyToRemove)) throw;
      recordList.length--;
      return true;
}
```

The `add` and `replace` functions also need to be updated.
For `add` function, we will first check whether the key exists, if it exists then throw. Then we will need to treat the new record, and set the index of record as `recordList.length`, since the index will start from 0 and index of the last record in `recordList` would be `recordList.length-1`. Since `.push()` returns the new array length, we can use it in concert with this step and reduce our gas cost a little, like this:

```
function add(address recordAddress, uint recordData) public
returns(bool success) {
      if(isRecord(recordAddress)) throw;
      recordStructs[recordAddress].record = recordData;
      recordStructs[recordAddress].index =
recordList.push(recordAddress) - 1;
      return true;
}
```

We will do similar things for `replace` function. It's simpler since we don't need to treat index of the record.

```
function replace(address recordAddress, uint recordData) public
returns(bool success) {
    if(!isRecord(recordAddress)) throw;
    recordStructs[recordAddress].record = recordData;
    return true;
}
```