Static Code Analysis: Creating and Managing a BlockChain.

Introduction

What are blockchains?
A blockchain is a series of blocks, each holding some kind of data and a hash value. The hash value of one block corresponds to the hash value of the previous block.

Lately, blockchains have been used as a form of decentralized cryptocurrency. Where each user of the blockchain has their own version of the blockchain. The data in the block is used to keep track of the transactions that occur and the hash value helps validate these transactions. As each user has their own copy, they can verify their copy with the other copies of the blockchain to see which one is the actual version (that is, which transactions actually occurred).

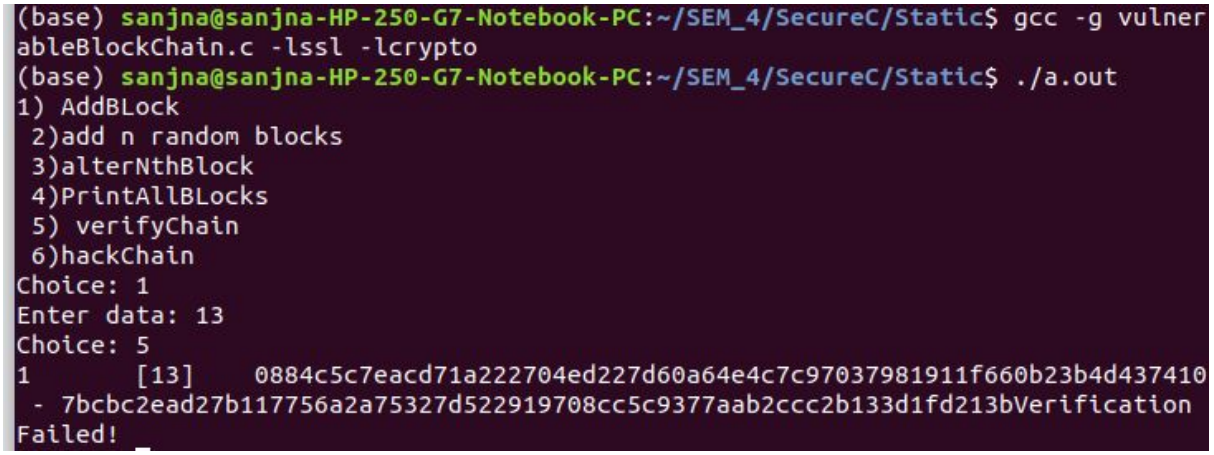
This decentralized nature, and the fact that hash values take a lot of computing power, add to the security of the blockchain.

About this application.
This application is used to make and maintain block chains. It uses the SHA256 to create a hash encryption value. The application has the following uses.

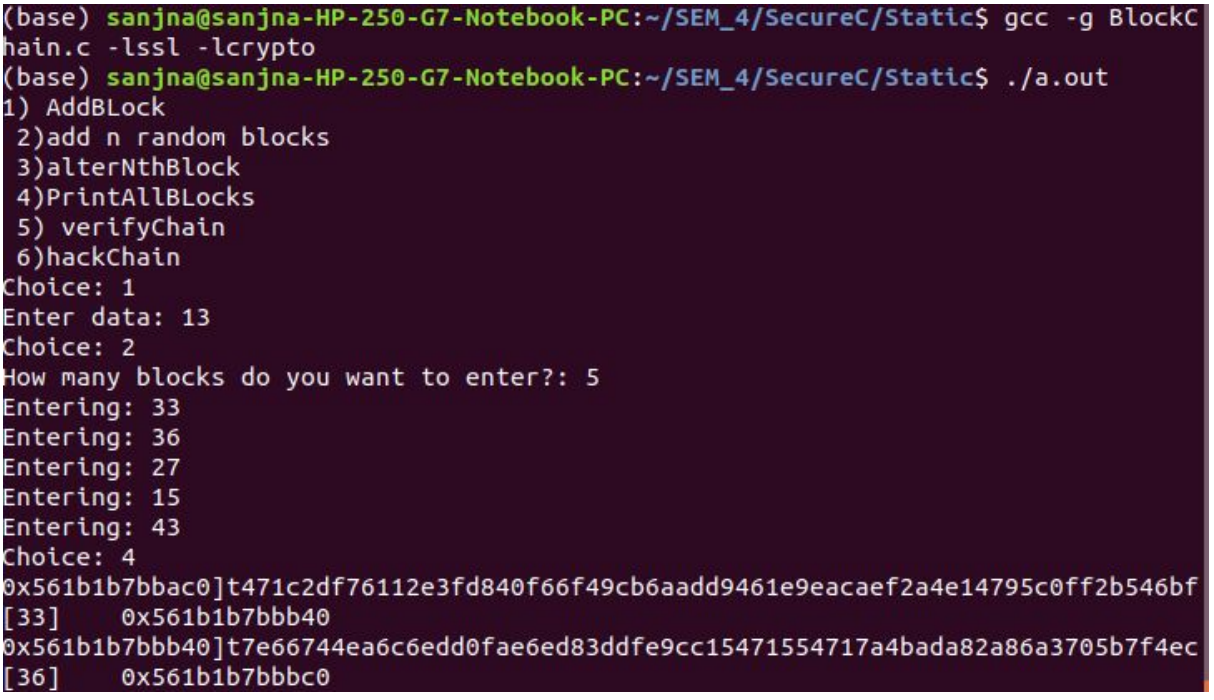
1. Add a single block to the block chain.
2. Add multiple blocks (each holding a random value)
3. Alter a block on the given chain.
4. Verify if the blockchain has been tampered with.
5. Fix any tampering that has occurred with the blockchain.

Execution

Vulnerable Execution:



Secure Execution:

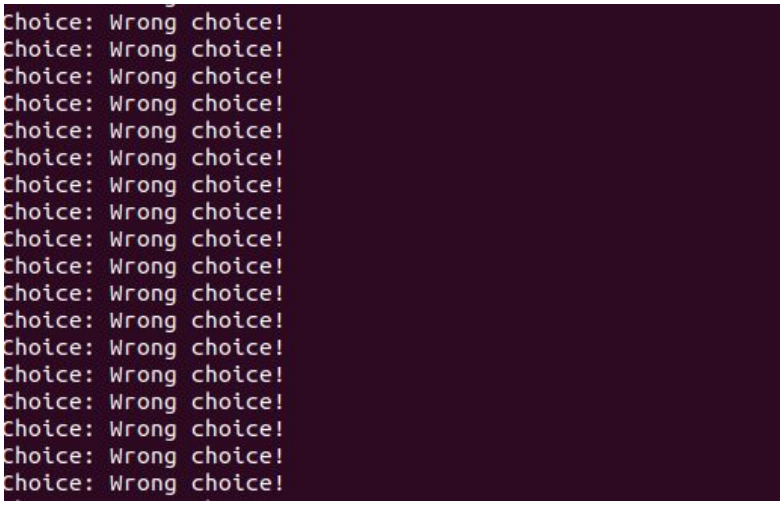


Vulnerabilities and Mitigations

Invalid user inputs

Vulnerability

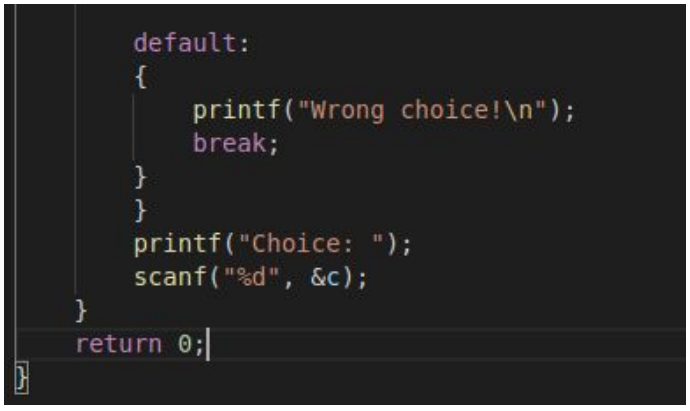
Our code breaks into an infinite loop when a user enters an invalid input. (anything that is not an int being supplied to the switch case)



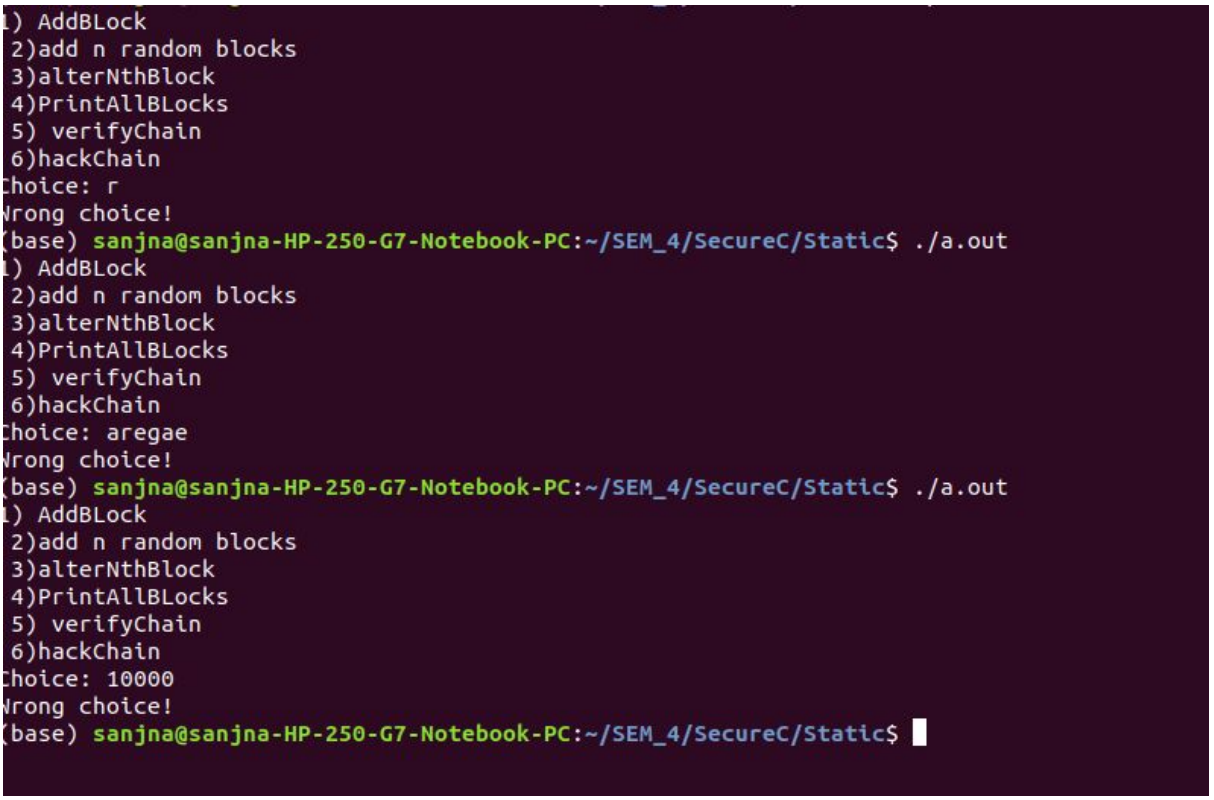
```
            break;
        case 7:
            break;
        default:
            printf("Wrong choice!\n");
            break;
        }
        printf("Choice: ");
        scanf("%d", &c);
    }
    return 0;
}
```

Mitigation

There are multiple causes for this. The first being, bad usage of Brackets.



```
        default:
        {
            printf("Wrong choice!\n");
            break;
        }
        }
        printf("Choice: ");
        scanf("%d", &c);
    }
    return 0;
}
```



Now our code terminates safely when it receives invalid inputs

Bad Input

Vulnerability

Function adblock expects an integer value, but when the user enters a character, it accepts the input. The block now holds an inaccurate value. We do not want this to happen.

Mitigation:

This problem also arises with our addBlock function.
The data in the block must be an integer. To ensure the user always puts in an integer, we use the following code.


```
printf("Enter data: ");
// scanf("%d", &n);
int num;
char term;
if (scanf("%d%c", &num, &term) != 2 || term != '\n')
{
    printf("failure\n");
    break;
}
addBlock(num);
break;
```

Memory Leaks

Vulnerability

This function behaves normally but on adding a block we lose memory as detected by valgrind.

The leak only occurs when we add the first block in. This indicates there is a problem while initializing the head. On further inspection, it becomes evident that we have forgotten to return.

```
1) AddBLock
 2)add n random blocks
 3)alterNthBlock
 4)PrintAllBLocks
 5) verifyChain
 6)hackChain
Choice: 1
Enter data: 12
==18387== Conditional jump or move depends on uninitialised value(s)
==18387==    at 0x10893F: addBlock (BlockChain.c:41)
==18387==    by 0x108F02: main (BlockChain.c:184)
==18387==
Choice: 1
Enter data: 12
Choice: 1
Enter data: 23
Choice: 0
==18387==
==18387== HEAP SUMMARY:
==18387==     in use at exit: 336 bytes in 7 blocks
==18387==   total heap usage: 9 allocs, 2 frees, 2,384 bytes allocated
==18387==
==18387== LEAK SUMMARY:
==18387==    definitely lost: 144 bytes in 3 blocks
==18387==    indirectly lost: 0 bytes in 0 blocks
==18387==      possibly lost: 0 bytes in 0 blocks
==18387==    still reachable: 192 bytes in 4 blocks
==18387==         suppressed: 0 bytes in 0 blocks
==18387== Rerun with --leak-check=full to see details of leaked memory
==18387==
==18387== For counts of detected and suppressed errors, rerun with: -v
==18387== Use --track-origins=yes to see where uninitialised values come from
==18387== ERROR SUMMARY: 3 errors from 1 contexts (suppressed: 0 from 0)
```

```
void addBlock(int data)
{
    if ((head = NULL))
    {
        head = malloc(sizeof(struct block));
        SHA256("", sizeof(""), head->prevHash);
        head->blockData = data;
    }
    struct block *currentBlock = head;
    while (currentBlock->link)
        currentBlock = currentBlock->link;
    struct block *newBlock = malloc(sizeof(struct block));
    currentBlock->link = newBlock;
    newBlock->blockData = data;
    SHA256(toString(*currentBlock), sizeof(*currentBlock), newBlock->prevHash);
}
```

Mitigation:

Use return;

```
void addBlock(int data)
{
    if (head == NULL)
    {
        head = malloc(sizeof(struct block));
        SHA256("", sizeof(""), head->prevHash);
        head->blockData = data;
        return;
    }
```

```
Choice: 1
Enter data: 12
Choice: 0
==17590==
==17590== HEAP SUMMARY:
==17590==     in use at exit: 48 bytes in 1 blocks
==17590==   total heap usage: 3 allocs, 2 frees, 2,096 bytes allocated
==17590==
==17590== LEAK SUMMARY:
==17590==    definitely lost: 0 bytes in 0 blocks
==17590==    indirectly lost: 0 bytes in 0 blocks
==17590==      possibly lost: 0 bytes in 0 blocks
==17590==    still reachable: 48 bytes in 1 blocks
==17590==         suppressed: 0 bytes in 0 blocks
==17590== Rerun with --leak-check=full to see details of leaked memory
==17590==
==17590== For counts of detected and suppressed errors, rerun with: -v
==17590== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Double Free.

Vulnerability

Valgrind detects us using memory that has not been allocated to us.

```
==19181== Invalid read of size 8
==19181==    at 0x108971: addBlock (BlockChain.c:42)
==19181==    by 0x108F57: main (BlockChain.c:185)
==19181==  Address 0x5b1b8e8 is 40 bytes inside a block of size 48 free'd
==19181==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==19181==    by 0x1089E7: addBlock (BlockChain.c:47)
==19181==    by 0x108FCD: main (BlockChain.c:197)
==19181==  Block was alloc'd at
==19181==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==19181==    by 0x10892B: addBlock (BlockChain.c:35)
==19181==    by 0x108FCD: main (BlockChain.c:197)
==19181==
```

```
    }
    struct block *currentBlock = head;
    while (currentBlock->link)
        currentBlock = currentBlock->link;
    struct block *newBlock = malloc(sizeof(struct block));
    currentBlock->link = newBlock;
    newBlock->blockData = data;
    SHA256(toString(*currentBlock), sizeof(*currentBlock), newBlock->prevHash);
    free(currentBlock);
}
```

Mitigation

The called function takes care of garbage collection. So we don't need to handle garbage collection in the calling function.

```
void addBlock(int data)
{
    if ((head = NULL))
    {
        head = malloc(sizeof(struct block));
        SHA256("", sizeof(""), head->prevHash);
        head->blockData = data;
    }
    struct block *currentBlock = head;
    while (currentBlock->link)
        currentBlock = currentBlock->link;
    struct block *newBlock = malloc(sizeof(struct block));
    currentBlock->link = newBlock;
    newBlock->blockData = data;
    SHA256(toString(*currentBlock), sizeof(*currentBlock), newBlock->prevHash);
}
```

```
==19933== HEAP SUMMARY:
==19933==     in use at exit: 48 bytes in 1 blocks
==19933==   total heap usage: 3 allocs, 2 frees, 2,096 bytes allocated
==19933==
==19933== LEAK SUMMARY:
==19933==    definitely lost: 0 bytes in 0 blocks
==19933==    indirectly lost: 0 bytes in 0 blocks
==19933==      possibly lost: 0 bytes in 0 blocks
==19933==    still reachable: 48 bytes in 1 blocks
==19933==         suppressed: 0 bytes in 0 blocks
==19933== Rerun with --leak-check=full to see details of leaked memory
==19933==
==19933== For counts of detected and suppressed errors, rerun with: -v
==19933== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Incorrect initialization

Vulnerability

Failing to initialize the count value to 0 will cause the alterNthBlock function to have undefined behaviour.

```
How many blocks do you want to enter?: 5
Entering: 33
Entering: 36
Entering: 27
Entering: 15
Entering: 43
Choice: 3
Which block do you want to alter?
2
Enter value: 12
Nth block does not exit
```

```
void alterNthBlock(int n, int newData)
{
    struct block *curr = head;
    int count;
    if (curr == NULL)
    {
        printf("Nth block does not exist!\n");
        return;
    }
```

Mitigation

To fix this, we must initialize the count to 0. Allowing us to actually change data and move through the chain safely.

```
How many blocks do you want to enter?: 5
Entering: 33
Entering: 36
Entering: 27
Entering: 15
Entering: 43
Choice: 3
Which block do you want to alter?
2
Enter value: 12
Before: 0x557cd5013b40]td765e9b9f803183737a9c3fa15d5b752db1b6d55cf0efe48b8d33c9bfac79684        [27]    0x557cd5013bc0

After: 0x557cd5013b40]td765e9b9f803183737a9c3fa15d5b752db1b6d55cf0efe48b8d33c9bfac79684 [12]    0x557cd5013bc0
```

Buffer OverFlow.

Vulnerability

The value returned by the malloc function is the destination for our memcpy(). But the size of the unsigned char has not been accounted for. This may seem small, but we are now at risk of a buffer overflow exploit. As our destination has less memory than our source.

```
unsigned char *toString(struct block b)
{
    unsigned char *str = malloc(sizeof(b));
    memcpy(str, &b, sizeof(b));
    return str;
}
```

Mitigation:

Multiplying by the size of unsigned char to ensure there is enough memory in the source.

```
unsigned char *toString(struct block b)
{
    unsigned char *str = malloc(sizeof(unsigned char) * sizeof(b));
    memcpy(str, &b, sizeof(b));
    return str;
}
```

Type Casting.

Vulnerability

Although this appears harmless, the hash we are using for this block chain (SHA256) is an unsigned char. Declaring the function this way, will cause unnecessary type casting, leaving us with more vulnerable code.

```
(base) sanjna@sanjna-HP-250-G7-Notebook-PC:~/SEM_4/SecureC/Static$ ./a.out
1) AddBLock
 2)add n random blocks
 3)alterNthBlock
 4)PrintAllBLocks
 5) verifyChain
 6)hackChain
Choice: 2
How many blocks do you want to enter?: 5
Entering: 33
Entering: 36
Entering: 27
Entering: 15
Entering: 43
Choice: 4
0x561a675dbac0]teb36c9ec89bed2cc9e8c47cf5677c1d50aec11e843fec1d6354f6a9366e0211f        [36]    0x561a675dbb40
0x561a675dbb40]t718834366ebd8c09f505a6fec1798b35c1ceafdbfd98a6f442d71b4b20cb0370        [27]    0x561a675dbbc0
0x561a675dbbc0]tc6cc75aad237773779b5ab71ac7b310dc9f85bfda897b3362ffc181714607a96        [15]    0x561a675dbc40
0x561a675dbc40]t98d8fb86ceb7da6244ab488014bd6406c454a1821a3aa83ceced8da269003812        [43]    (nil)
Choice:
3
Which block do you want to alter?
4
Enter value: 23
Before: 0x561a675dbc40]t98d8fb86ceb7da6244ab488014bd6406c454a1821a3aa83ceced8da269003812        [43]    (nil)

After: 0x561a675dbc40]t98d8fb86ceb7da6244ab488014bd6406c454a1821a3aa83ceced8da269003812 [23]    (nil)
```

```c
char *toString(struct block b)
{
    char *str = malloc(sizeof(b));
    memcpy(str, &b, sizeof(b));
    return str;
}
```

Mitigation:

It is safer to explicitly define the char as unsigned.

```c
unsigned char *toString(struct block b)
{
    unsigned char *str = malloc(sizeof(unsigned char) * sizeof(b));
    memcpy(str, &b, sizeof(b));
    return str;
}
```

Integers without limits

Vulnerability

The data inside the block is stored in an integer. This could potentially lead to undefined behaviour and faulty data as integers don't have very well defined limits.

```
struct block
{
    unsigned char prevHash[SHA256_DIGEST_LENGTH]; //unsigned char array
    int blockData;
    struct block *link;
} * head;
```

Mitigation :

Including the <limits.h> header file

```
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <limits.h>
```

Referencing Freed Memory

Vulnerability

While entering data into the blockchain, everything appears to function normally. But when we try to access this data, we have undefined behaviour. On trying to verify the chain, it told us there were no blocks. On trying to print all blocks, it caused a segmentation fault.
This happened because we freed all pointers that can access the heap.

```
1) AddBLock
 2)add n random blocks
 3)alterNthBlock
 4)PrintAllBLocks
 5) verifyChain
 6)hackChain
Choice: 1
Enter data:
12
Choice: 2
How many blocks do you want to enter?: 5
Entering: 33
Entering: 36
Entering: 27
Entering: 15
Entering: 43
Choice: 5
BlockChain is empty Please try again after adding in some blocksChoice: 4
Segmentation fault (core dumped)
```

```
head->blockData = data;
free(head);
head = NULL;
```

Mitigation

We cannot free head as it needs to be used by other functions.
Don't free pointers that still need to be used.

```
SHA256("", sizeof(""), head->prevHash);
head->blockData = data;
return;
```

00722_Sanjna C

Safe Exits

Vulnerability

Although not a vulnerability, it is always good programming practice to make sure a user (or another developer) knows why a program has terminated.

Mitigation

Added in error messages on the off chance that or dynamic memory allocation failed or any other part of the program fails.

```
printf("malloc failed: No more memory. Goodbye.\n");
exit(1);
```