

Analyzing Code for Security Defects

How to review large code base for security defects.



Security
compass

Defense In Depth Today

- » Firewalls
- » Separation of Networks (DMZ)
- » Network / Host Assessments
- » Bastion Hosts / Hardened Builds
- » Vulnerability Scanning
- » Product Review
- » Code Review

Methodology of Reviewing Large Code Set

1. Threat Model
2. Cursory Review of Code
3. Separation of Code [Standard Model & Application Architecture]
4. Maintain code notes with reviewer name
5. Detailed Code Analysis
6. Common list of issues to review [C/C++ Language Specific]

Threat Analysis

- » Overview
- » What Is Threat Analysis
- » When Threat Analysis
- » Why Threat Analysis
- » Who Threat Analysis
- » How Threat Analysis
 - Collecting Information / Decomposing Application
 - Modeling the System
 - Analysis to Determining Threat

C/C++ Language Specific

» Termination

- Null Termination
- Conditional Termination
- Premature Termination

» Validation

- Exported Functions
- Command Line
- String Formatting

» Calculations

- Division
- Signed
- Integer
- Off by one / few

Threat Modeling

What is Threat Modeling ?

- » Threat modeling is an organized method of attacking an application. It can be considered as a systematic method of finding security issues in application.
- » Threat Modeling can be viewed as a reversal of roles, where by a developer attempts to think as an attacker to determine possible compromises/threats in his application.

What is Threat Modeling ?

- » Hackers/Attackers have been threat modeling for a while now. They haven't used the terminology "Threat Modeling"
- » Security Groups have formalized the process to help developers and testers better understand the different threats that might exist in an application.
- » Threat Modeling helps educate developers and testers about potential vulnerabilities that might pop up in the application that might not have been properly mitigated.
- » It teaches constructive paranoia not only to project managers but also to developers.

Why Threat Model ?

Threat Modeling can help –

- » Develop countermeasures for those threats / risks
- » Weigh each threat (assign value to them)
- » Produce a secure application
- » Review code for security defects in large code base
- » Understand risks & threats to the application


Who should Threat Model ?

- » Developers / PM / Business Dev / Security Group / Any One else
- » Business Dev
 - Explain the goal of the application. (so the main goal is still met).
- » PM / Application Architect
 - Provide Data Flow Diagram / Application architecture and explain the app path in detail
 - Help understand why a particular path is chosen to develop the application
- » Developers
 - Approximate time frame on the application dev process
 - Understand potential threats.
- » Security Group
 - Point out different points of weakness (Risks & Threats).

When should you threat Model ?

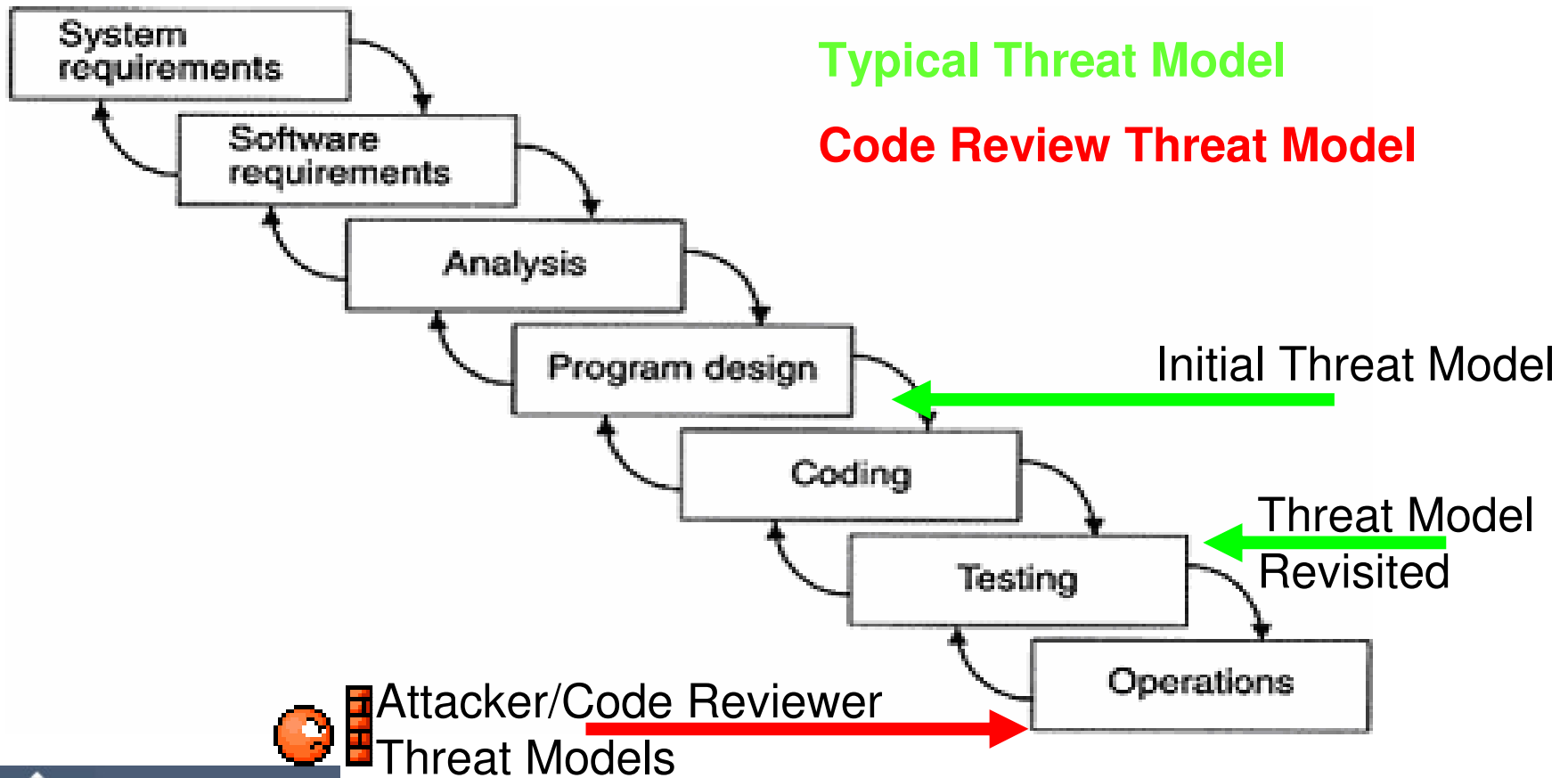
- » Most suitable to perform threat model after the application architecture has been developed (After design stage)
- » Threat Modeling must be revisited at least once when the application is in Alpha release (Before format testing starts).
- » Ideally Threat Modeling must be performed every time an application is tested for any change (Functionality/Security/Any other fix/Upgrade).

When do organizations threat Model (Typically)

- » Performed after application is vulnerable (the stage after the application  has been released).
- » Attackers also perform threat modeling at this stage.

Note: Every application that is being developed or has already been developed should be threat modeled, even if the application is being built for internal use only.

SDLC - Waterfall Model



How to Threat Model (The Process)

- » Collecting Information about the application (background why the application is built etc).
- » Decomposing Application / Modeling the System (break the application down into reasonably separate chunks either by functionality or connectivity).
- » Analysis to Determine Threats (perform a walk through to determine the different locations of issues).

Collecting Information

(process step 1)

» Collecting Information

- How the application is intended or not intended to be used in deployment
- Any dependencies that exists (external / inter process dependencies / account level dependencies / application requirement example: mail server etc).

Decomposing

- » Decomposing Application / Separating the application into reasonable chunks
 - Chunks either based on Application Architecture / functionality (specially if performing code review, help code reviewer).
 - Chunks based on individual components (Entry Points, Trust Points etc)

Decomposing - Application Architecture

- » Authentication
- » Authorization
- » Session Management (web)
- » User Management
- » Cryptography
- » Data Validation
- » Error & Exception Handling
- » Event Logging

Decomposing – Individual Components

- » Entry Points
 - Identify all entry points to the application
 - Network accessible (RPC / TCP / Web Services etc)
 - Locally accessible (Registry / LPC / File / command line / environment variables etc)

- » Trust Level
 - Identify different trust boundaries

Decomposing - Modeling the System (DFD)

- » Data Flow Diagrams
 - Drawing data flow diagrams or other models to visual represent the application is called modeling the system. Using DFD makes it easy to understand the application and data flow of the application, thus making it easy for anyone reviewing the application to get a better understanding of the application.

- » A DFD
 - A graphical representation showing communication between objects
 - Describe activities that process data
 - Show how data flows through a system
 - Show logical sequence of associations and activities

Analysis to Determine Threats

- » Identify threats and create attack scenarios on the basis of the DFD (is the single biggest challenge).
- » Threats is not the same as vulnerabilities:-
 - Threats – possibly dangerous
 - Vulnerability – susceptible to attack
- » Analysis of these threats can determine if a threat is mitigated or can result in a vulnerability.

Definitions

Dictionary.com

- » **Threats:** One that is regarded as a possible **danger**; a menace.
- » **Risks:** The possibility of suffering harm or **loss**; danger.
- » **Vulnerabilities:** Susceptible to **attack**.

Writing Secure Code 2

- » **Threats:** A malicious entity that might try to attack. A Threat does not **constitute** a vulnerability.
- » **Risks:** The chance of something going wrong.
- » **Vulnerabilities:** A weakness in a system that can be exploited. A Vulnerability exists when there is a Threat that goes unmitigated.

Assigning Value to Threat

Assigning Value to Threats.

- » The second biggest challenge is assigning value to threats.
(Note: First was Identify threats and create attack scenarios on the basis of the DFD)

- » A Model used and developed at MS is the DREAD model.

DREAD (WSC) (1-10) / 5

- Damage Potential
- Reproducibility
- Exploitability
- Affected Users
- Discoverability

Application Architecture – Threat Model

(Propose)

- » Authentication
- » Authorization
- » Cryptography
- » Data Validation
- » Error & Exception Handling
- » Logging

(Low-1, Mid-2, High-3)

Low (Intranet)

Mid (Internet Non Critical – no PII or other critical data is being pulled)

High (Internet Critical Data)

Application Architecture – Threat Model

(Propose)

- » Modeling the System
 - Decompose it on the basis of the architecture
 - Entry Points in each location
 - Trust Levels

- » Assign Value Depending on location of each Architecture
 - Authentication 1 2 3
 - Authorization 1 2 3
 - Crypto 1 2 3
 - Data Validation 1 2 3
 - Error 1 2 3
 - Logging 1 2 3

Application Architecture – Threat Model

(Propose)

- » Min is > 8 Low Risk
- » Mid if > 12 Medium Risk
- » Max if > 18 High Risk

- » Total if within range then considered high risk (13-18) , should follow best security practice and need to fix bugs immediately.
- » Total if within range then considered Medium risk (7-12) , should follow best security practice and need to fix bugs at reasonable quick. (next patch release)
- » Total if within range then considered high risk (>6) , should follow best security practice and need to fix bugs. (next point release)

Threat Model Checklist

- ✓ Every Application Should Have A Threat Model At Least Once
- ✓ Every Threat Must Be Analyzed
- ✓ A Threat Value Must Be Assigned to Every Threat
- ✓ Bugs Must Be Fixed Based On The Threat Value

How Do You Review Code?

Methodology

- ☑ Step 1) Threat Model
- ☑ Step 2) Everyone Read The Code
(Cursory review to understand contents of each file and global vars)
- ☑ Step 3) Break Code Into Separate Chunks / Same as DFD
(Individual can review their sections)
- ☑ Step 4) Maintain code notes with reviewer name
- ☑ Step 5) Detailed Code Analysis
- Step 6) Common list of issues to review [C/C++ Language Specific]

Reviewing Code


C/C++ Some of the Common Issues

- » Commonly seen issues while reviewing code.
- » Many Issues Exists, we will cover three such major topics, which lead to vulnerabilities in applications.
 - Termination Issues
 - Validation Issues
 - Calculation Issues

» Termination

- Null Termination and strlen
- Null Termination and strncpy
- Conditional Termination
- Premature Termination

Null Termination “\0” and strlen

```
void foo (char* input)
{
    char* output = NULL;
     output = (char*)malloc((strlen(input) ) * sizeof(char));
        if(output != NULL) { //do processing...
        }
    }
    output = (char*)malloc((strlen(input) + 1) * sizeof(char));
    //The +1 is for the terminating NULL
```

When allocating memory for a string always bear in mind the fact that the *strlen* returns the length of a string excluding the terminating NULL. Hence, it is necessary to explicitly allocate for this terminator as shown.

NULL Termination “\0” and strlen

- » If strlen is not increased by one then, string operations would not perform as expected.
- » When copying string characters manually in a loop, it is important to NULL terminate them at the end. This issues is seen when a programmer attempts to handle the length properly, however, a string can be created without a trailing NULL. This often happens when using a *strncpy* type function operation.

NULL Termination “\0” and strncpy

- » For example, this code copies foo into buff without checking for NULL Termination or adding it in. When strlen tries to determine the length, it will read past the end of buff searching for NULL Termination that doesn't exist.

```
char buff[10];  
strncpy(buff, foo, sizeof(buff));
```



```
if (strlen(buff) <= 10)
```

- » buff is left unterminated if strlen(buff) equals 10

- » MSDN states:

“The **strncpy** function copies the initial *count* characters of *strSource* to *strDest* and returns *strDest*. If ***count*** [in above example: sizeof(buff)] **is less than or equal to the length of *strSource*** [in above example: buff] , **a null character is not appended** automatically to the copied string. If *count* is greater than the length of *strSource*, the destination string is padded with null characters up to length *count*.”


Conditional Termination

```
{
    int StringLength;
    size_t index;
    int BufferLength=20;
    TCHAR *Buffer=argv[1];
    printf("%s", Buffer);
    index = strlen(Buffer);
    printf ("%d", index);
    while (index < BufferLength && Buffer[index] != '\0')
        index++;
    StringLength = strlen(Buffer);
    // Hint: What happens if index < BufferLength & thus the first Clause completes the
    // while loop
}
```

Conditional Termination

- » The loop seems to be attempting to check that the buffer is properly NULL-terminated without overflowing the end of the buffer, but the statement immediately following assumes that the terminator was found, and thus the second condition is what terminated the while loop.
- » However, if the first clause is what fulfilled the termination condition, the strlen call will read past the true end of the buffer.
- » It is therefore important to ensure that the logic checks for all conditions including failures.

Premature Termination

```
int main(int argc, char* argv[])  
{  
    int a = 1;  
     if (a==2);  
    {  
        printf ("hello world\n");  
    }  
    return 0;  
}
```

Premature Termination

- » In the C and C++ programming languages, ; terminates statements
- » Premature termination of the if-statement causes the subsequent statement to be executed always.

» Validation

- Exported Functions
- Command Line
- String Formatting

Validation – Command Line

```
int copy(char* input) {  
    char var[20];  
    🚫 strcpy (var, input);  
    return 0;  
}  
  
int main(int argc, char* argv[]){  
    copy(argv[1]);  
    return 0;  
}
```

Validation – Reading from network

```
void pr( char *str)
{  char buf[2000]="";
  🚫 strcpy(buf,str);
}
while( bytesRecv == SOCKET_ERROR )
{//receive the data that is being sent by the client max limit to 5000 bytes.
bytesRecv = recv( clientSocket, Message, 5000, 0 );
    if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
    {
        printf( "\nConnection Closed.\n");
        break;
    }
}
```

Validation – Exported Functions

- » Any public function, for example, an exported function from a dynamic or statically linked library or a function accessible via an RPC interface, is vulnerable to attack via it's parameters.

//filenm is an input filename, len is the length

```
__declspec(dllexport) int expfunc (char *Filenm, size_t Len)
{
    char szCopy[MAX_PATH]; //260
    strncpy(szCopy, Filenm, Len);
    return 0;
}
```

- » In the example if len is greater than MAX_PATH then Copy will not be large enough to accommodate the data being copied. All public functions should always validate all the input passed to them.

Validation – String Formatting

```
void main()
{
    char buf[20]="";
    strncpy(buf,argv[1],20);
    printf(buf);
}
```

The function prints the data that is provided as an argument to the function using printf function.

The function however doesn't format the data. what would happen if argv[1] contained "%0.X"

» Calculation

- Division
- Signed
- Integer
- Unicode
- Off by one / few

Calculation - Division

```
int divide(long x)
{
    long y;
    🐛 y = (4096 / (x / sizeof(long))); // Hint: What is the lowest value of x
    printf("%d\n", y);
    return y;}

int main(int argc, char* argv[])
{
    int x;
    sscanf(argv[1], "%d", &x);
    divide(x);
    return 0;
}
```

What would happen if x is less than 4

Calculation - Division

- » If the value of x is larger than the *sizeof(long)* (usually 4 bytes) the program would function properly, however when x is less than 4, for instance 1, the value of r is $(4096) / (1/4)$ which would result in division by zero, since integer division of $1/4$ results in 0.
- » Hence, special care should be taken in algorithms that require calculation to be performed with either user supplied variables or derivatives of user supplied variables to ensure that there is no possibility of division by zero.
- » While performing division on any values, ensure that the division is checked, even if the caller is a trusted source.

Calculation - Integer

```
int main(int argc, char **argv)
{
    int i;
    🐛 u_int malloc_size, size;
    char *data;
    size = atoi(argv[1]);
    malloc_size = size * 4; // hint: what is the data type of size compare it to malloc_size
    (u_int)
    data = (char *)malloc(malloc_size)
    if(data != NULL)
        {printf("%p\n", data);
        printf("malloced %d bytes of data\n", malloc_size);
        for(i = 0; i < size; i++)
            data[i] = argv[2][i];
        }
}
```


Calculation - Integer

- » The data type of a variable defines the maximum / minimum value allowed for it, based on the number of bytes it occupies.
- » What a user can do is pass to the program a large integer. When the program calculates: $malloc_size = size * 4$, the *malloc_size* variable will be overflowed and truncated.
- » For instance, if a variable is declared as *short*, the maximum value the variable can store is 32767 (16 bits or 2 bytes long) while the minimum value is -32767. Hence, if the value stored exceeds 32767 it would lead to corruption of data.

Calculation - Signed

```
int main(int argc, char **argv)
{
    🗨️ int size;
    char data[1024];
    size = atoi(argv[1]); // Hint: what would happen if you provided int+1
    if(size > 1024)
        return;
    memcpy(data, argv[2], size);
}
```

Calculation - Signed

- » The example attempts to implement a check to prevent integer overflow - *if (size > 1024) -*, however there is a subtle flaw, in that it uses a signed integer for the size variable. Thus, the check can be defeated by specifying a negative number. Ensuring the right data type is used could prevent this error e.g. `u_int size;`
- » Signed issues occur when a signed variable is interpreted as an unsigned variable. This commonly occurs in cases where casting is used to convert from signed to unsigned types and vice versa.
- » While creating loops as a best practice always use unsigned integers.

Calculation: Unicode

```
void GetData(char *fromData)
{
    WCHAR toData[10]; //means 20 bytes
    🐛 MultiByteToWideChar(CP_ACP, 0, fromData, -1, toData, sizeof(toData));
    // Hint: what is sizeof(toData) going to return
}


int main(int argc, char * argv[])
{
    GetData("0123456789");
    return 0;
}

MultiByteToWideChar(CP_ACP, 0, fromData, -1, toData, sizeof(toData)/sizeof(WCHAR))
```


Calculation – Off by one/few

» C/C++ arrays start at 0

```
int buff[100];  
buff[100] = 100;
```




```
char buff[MAX_PATH];  
buff[sizeof(buff)] = 0;
```



» should be `buff[sizeof(buff) - 1]`

```
int buff[SIZE]; for (int j = 0; j <= SIZE; j++)
```



» should be `< SIZE` and not `<= buff[j] = 0;`

Methodology

- ☑ Step 1) Threat Model
- ☑ Step 2) Everyone Read The Code
(Cursory review to understand contents of each file and global vars)
- ☑ Step 3) Break Code Into Separate Chunks / Same as DFD
(Individual can review their sections)
- ☑ Step 4) Maintain code notes with reviewer name
- ☑ Step 5) Detailed Code Analysis
- ☑ Step 6) Common list of issues to review [C/C++ Language Specific]

Methodology

- ✓ Threat Model
- ✓ Do a Two Pass Code Review
- ✓ Break Code Review Into Major Separate Sections (As in TM)
- ✓ Create Major Worksheet Which Consists Of Global Variables, Global Functions and Classes.
- ✓ Code Reviewer Should Append His Own Comments About The Code.

Questions

Nish[a t]securityCompass.com

Visit us at

www.SecurityCompass.Com