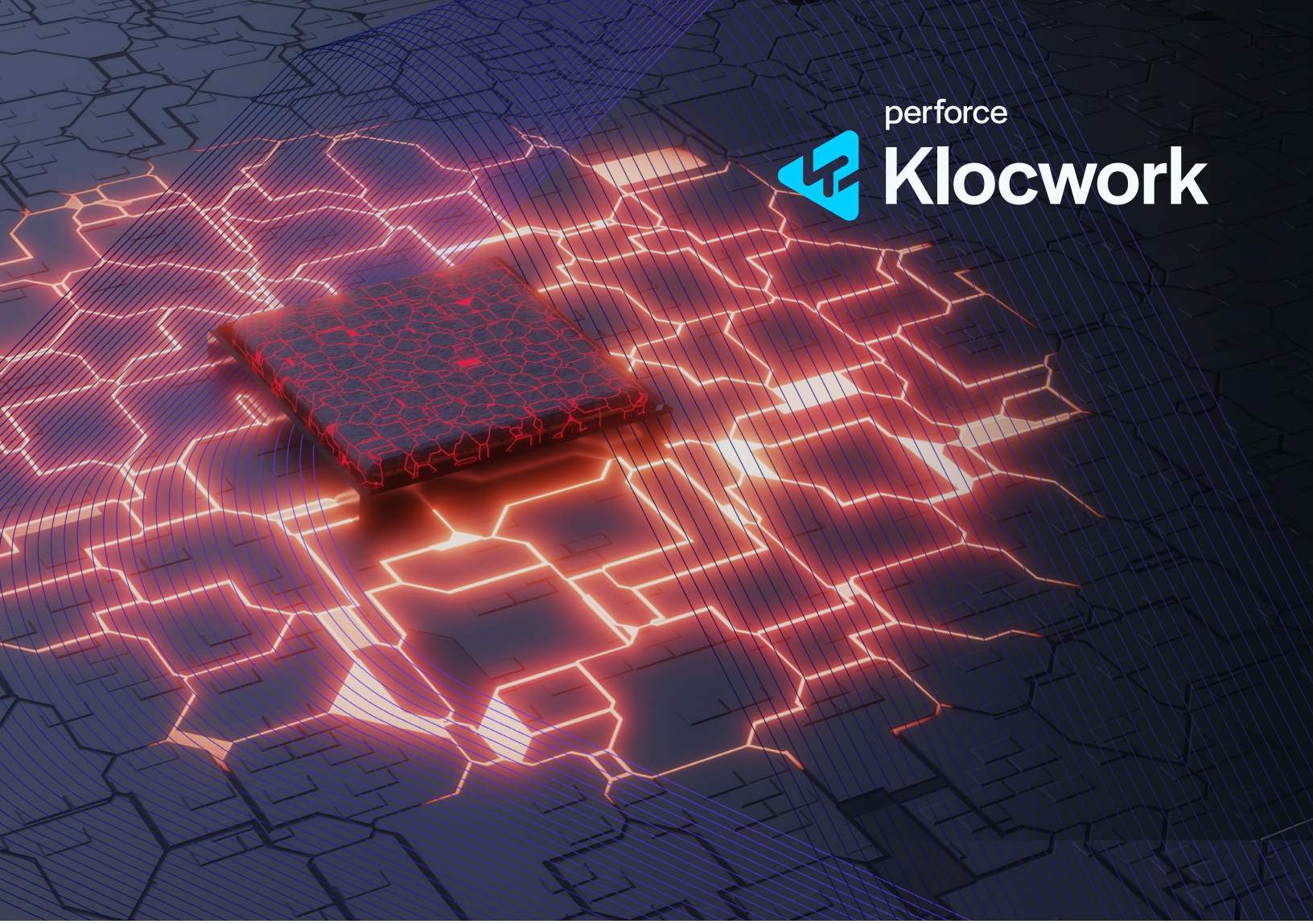

WHITE PAPER

Top 10 Embedded Software Cybersecurity Vulnerabilities

Introduction

Trying to build embedded software that's safeguarded against cybersecurity vulnerabilities can be a difficult and time-consuming task. According to embedded software industry experts, nearly 90% of all detected security holes can be traced back to just 10 types of vulnerabilities.

In this paper we'll explore the top 10 most common cybersecurity vulnerabilities, provide examples from actual source code, and look at what changes can be made to coding style or processes to avoid them.

Contents

Vulnerability 10: Numeric Errors

Numeric errors can refer to several different categories of problems, including:

- Wrap around errors.
- Improper validation of array index.
- Integer overflow.
- Incorrect byte ordering.
- Incorrect conversion between numeric types.
- Numeric errors make up 3 percent of all vulnerabilities.
- Incorrect calculation.

A common place for numeric error vulnerabilities is in math calculations. Also, if data is overflowing from an external source, it's subject to this kind of a vulnerability.

Example of a Numeric Error: Qt

A perfect example of a numeric error is a bug in the bitmap, or BMP file decoder, in Qt. As seen in figure 1, the code doesn't properly calculate the masks used to extract the color components like red, green, blue, and alpha components of the image.

```
} else if (comp == BMP_BITFIELDS && (nbits == 16 || nbits == 32)) {
    red_shift = calc_shift(red_mask);
    red_scale = 256 / ((red_mask >> red_shift) + 1);

    green_shift = calc_shift(green_mask);
    green_scale = 256 / ((green_mask >> green_shift) + 1);

    blue_shift = calc_shift(blue_mask);
    blue_scale = 256 / ((blue_mask >> blue_shift) + 1);

    alpha_shift = calc_shift(alpha_mask);
    alpha_scale = 256 / ((alpha_mask >> alpha_shift) + 1);

} else if (comp == BMP_RGB && (nbits == 24 || nbits == 32)) {
```

Figure 1: Example of Qt failure

If a bad image is created in just the right way, the software that is creating that image will crash. The example in figure 1 may have taken a little bit longer to be recognized because it isn't a JPEG or other common image file.

As the code in figure 1 shows, there are shift and scale values that are being calculated from the bitmap that is being loaded, and in this example, 256 is being divided by a calculated value of the color mask shifted by a shift value, and then adding one. This will make the system crash if the mask value is a bunch of 1s and the shift value is a bunch of 0s because 256 can't be divided by zero.

Though some people may claim that this situation would never really happen, that isn't the point. The main point is that security is very difficult because it's not just about coding correctly, it's also about thinking maliciously and coding defensively. It's no longer acceptable to simply code for good circumstances, developers must now code defensively for bad circumstances.

How to Solve Numberic Error Vulnerabilities in Qt

Figure 2 shows how to solve the vulnerability in Qt, by adding code that checks every time a denominator might come out as equaling zero — the image is flagged as bad and the program bails on that set of code. This will ensure that the code never produces a zero answer because the code is tested every single time to ensure that the program bails out on bad images.

```
} else if (comp == BMP_BITFIELDS && (nbits == 16 || nbits == 32)
    red_shift = calc_shift(red_mask);
    if (((red_mask >> red_shift) + 1) == 0)
        return false;
    red_scale = 256 / ((red_mask >> red_shift) + 1);
    green_shift = calc_shift(green_mask);
    if (((green_mask >> green_shift) + 1) == 0)
        return false;
    green_scale = 256 / ((green_mask >> green_shift) + 1);
    blue_shift = calc_shift(blue_mask);
    if (((blue_mask >> blue_shift) + 1) == 0)
        return false;
    blue_scale = 256 / ((blue_mask >> blue_shift) + 1);
    alpha_shift = calc_shift(alpha_mask);
    if (((alpha_mask >> alpha_shift) + 1) == 0)
        return false;
    alpha_scale = 256 / ((alpha_mask >> alpha_shift) + 1);
} else if (comp == BMP_RGB && (nbits == 24 || nbits == 32)) {
```

Figure 2: Example of Qt fix

Figure 2: Example of Qt fix

It's impossible to truly be sure where images are coming from. The assumption is often made that logos and images are coming from a safe location, but the code could be pulling images from Bluetooth or web-based apps without developers having visibility into this.

To help with numeric errors you should, enforce, regulate, and pay attention to:

- Architecture and design guidelines.
- Implementation guidelines.
- Manual analysis.
- Automated static analysis.
- Automated dynamic analysis.

Security needs to be considered a high priority through the entire build phase. It's important to remember that no matter how small or inconsequential an error seems,it could lead to a malicious attack.

Vulnerability 9: Cryptographic Issues

Cryptographic issues are weaknesses related to the use of cryptography, and baffle many developers. There are a lot of things that could mess up cryptography, including:

- Missing encryption of sensitive data.
- Key management errors.
- Missing required cryptographic step.
- Inadequate encryption strength.
- Use of broken or risky cryptographic algorithms.
- Use of reversible one-way hash.
- Not properly using random initialization.
- Improper use of RSA algorithm.

Example of Cryptographic Issues: Using OpenSSL

OpenSSL was made famous a short time ago for Heartbleed, which is a separate issue than what we cover later in this paper. Another issue with OpenSSL is when it doesn't ensure that the pseudo-random number generator gets seeded before it proceeds with the handshake.

What this basically means is that there isn't sufficient information to ensure that the output isn't predictable when it goes through the algorithm. This means that the space that is needed to test for possible keys against encrypted data is drastically reduced and attackers will therefore be able to break the encryption eventually. The hacking of Wi-Fi WPA encryption is a good example because the way those numbers are seeded, regardless of the WPA security, make it hackable due to the consistent time around startup.

As seen in figure 3, the command in question is a piece of the client buffer in the loop, where it says if a condition happens then fill the random buffer. The problem with this is that the fill_hello_random is not being called if that condition is non-zero. This means that there are paths through that code where it won't be a random buffer when the handshake is started, which can lead to predictable results.

```
int ssl3_client_hello(SSL *s)
{
...
    p = s->s3->client_random;

    /*
     * for DTLS if client_random is initialized, reuse it, we are
     * required to use same upon reply to HelloVerify
     */
    if (SSL_IS_DTLS(s)) {
         size_t idx;
         i = 1;
         for (idx = 0; idx < sizeof(s->s3->client_random); idx++) {
              if (p[idx]) {
                   i = 0;
                   break;
              }
         }
     } else
         i = 1;

    if (i)
         ssl_fill_hello_random(s, 0, p, sizeof(s->s3->client_random));

     /* Do the message type and length last */
      d = p = ssl_handshake_start(s);
```

Figure 3: Example of OpenSSL failure

The Fix: Always Call a Randomizing Function

As seen in figure 4, there should always be a test to make sure that the value is acceptable, and then the randomizing function should always be called. So now if a zero flag comes up it can be aborted, which ensures that the handshake can't start without having a randomizing buffer — which is crucial.

```
int ssl3_client_hello(SSL *s)
{
...
    p = s->s3->client_random;

    /*
     * for DTLS if client_random is initialized, reuse it, we are
     * required to use same upon reply to HelloVerify
     */
    if (SSL_IS_DTLS(s)) {
        size_t idx;
        i = 1;
        for (idx = 0; idx < sizeof(s->s3->client_random); idx++) {
            if (p[idx]) {
                i = 0;
                break;
            }
        }
    } else
        i = 1;

    if (i && ssl_fill_hello_random(s, 0, p,
                                   sizeof(s->s3->client_random)) <= 0)
        goto err;

    /* Do the message type and length last */
    d = p = ssl_handshake_start(s);
```

Figure 4: Example of OpenSSL fix

Vulnerability 8: Code Injection

Code injections are something that affect interpreted environments such as PHP, making this a large vulnerability among the website development community. Despite this vulnerability being largely seen in website development, it's still very much present when it comes to infotainment systems and other complicated in embedded systems.

Because there is usually a scripting engine in place to take care of a lot of the service startup, there is usually an extra vector that can be attacked. This can also affect black box component containing interpreters that developers might be unaware of, like an engine.

Example Code Injection: Windows RT

One example is Windows RT, which is interesting because this isn't open source code. This proves that even though something may be proprietary, it doesn't mean that it isn't open to vulnerabilities. This is important to know when doing integration of these technologies into code. Something as innocent seeming as being able to display fonts could be the reason why code has become vulnerable.

How to Fix Vulnerabilities in Proprietary Code

Design review, manual analysis, and automated static analysis can all help remediate vulnerabilities, but understand that securing proprietary code can be challenging.

From a design standpoint developers need to be aware of all black box components and that all of them are up-to-date, and shouldn't assume that everything is within an internal system. It's important that developers pay special attention to items that may be pulling from a website.

Static code analysis (SCA) can help in this realm by identifying the use of unsafe data as it flows through the system. This is where developers will likely catch an injection-based attack. Manually cleaning any externally acquired information will also help keep systems secure.

It's important to keep in mind all of the code that is coming into an ecosystem; assume that there are vulnerabilities in multiple stages of the software development lifecycle.

Vulnerability 7: Code

It may seem odd to have code as its own vulnerability, but what we're talking about is anything that doesn't fall into a specific category — so consider this a catchall vulnerability. This can include things like:

Mismanaging passwords, storing plaintext passwords, hardcoded passwords.

- Improper handling of API contracts.
- Improper or absent error handling.
- Improperly handling time and state.
- Code errors make up 4.4 percent of all vulnerabilities.
- Code generation issues.

It's important that developers don't make their own encryption — it isn't worth it. It's very easy to reverse engineer and that has left many people frustrated and embarrassed when it doesn't behave how they anticipated.

Code Vulnerability: Chrony (NTP)

The example we have for this catchall of vulnerabilities is chrony (NTP) where it's not initializing the last "next" pointer when saving unacknowledged replies to command requests. What ends up happening is the execution of arbitrary code or at least allows the creation of a denial of service attack.

In figure 5 is a code snippet to explain the vulnerability. The sample shown is very subtle, and might actually require some studying before seeing why there is a vulnerability. Basically, the issue in the example is that this a very long and complicated test, and that the logical OR in the code causes an invalid time stamp when it comes through with an `issue_token` of one (1). This means that a token is going to be allowed to be issued without properly examining the request.

```
valid_ts = 0;

if (auth_ok) {
    struct timeval ts;

    UTI_TimevalNetworkToHost(&rx_message.data.logon.ts, &ts);
    if ((utoken_ok && token_ok) ||
        ((ntohl(rx_message.utoken) == SPECIAL_UTOKEN) &&
         (rx_command == REQ_LOGON) &&
         (valid_ts = ts_is_unique_and_not_stale(&ts, &now))))
        issue_token = 1;
    else
        issue_token = 0;
} else {
    issue_token = 0;
}
```

Figure 5: Example of chrony failure

How to Fix Fix Chrony in the Code

Fundamentally, the fix found in figure 6 is making the control logic much easier to see and much clearer. This is the way that code should always be designed from the beginning so that code doesn't become overcomplicated. Though it may seem boring, sometimes coding by the book will save developers from a vulnerability.

```
valid_ts = 0;
issue_token = 0;

if (auth_ok) {
    if (utoken_ok && token_ok) {
        issue_token = 1;
    } else if (rx_command == REQ_LOGON &&
               ntohl(rx_message.utoken) == SPECIAL_UTOKEN) {
        struct timeval ts;

        UTI_TimevalNetworkToHost(&rx_message.data.logon.ts, &ts);
        valid_ts = ts_is_unique_and_not_stale(&ts, &now);

        if (valid_ts) {
            issue_token = 1;
        }
    }
}
```

Figure 6: Example of chrony fix

Be sure to use well-identified coding patterns, as tricky coding doesn't do anyone any good. Also create consistent API contracts and track the use and storage of encrypted data passwords.

Keep an eye out for unclean code, as it often points to poor quality and potential security issues. Unclean code itself isn't always the vulnerability, but unclean code can be an indicator of where there was no methodical process. And be sure to handle all errors so there aren't any surprises.

Good coding means good security. Coding style guides may not be the most popular, but rest assured that they exist to keep vulnerabilities at bay.

Vulnerability 6: Resource Management Errors

Resource management errors can refer to a number of things, including:

- Improper management of system resources.
- Uncontrolled resource consumption.
- Transmission of private resources into new sphere.
- Improper resource release or shutdown.
- Asymmetric resource consumption.
- Resource locking.
- Double-free, use after free.
- Insufficient resource pool.
- Free of non-heap memory.

Though these problems are much more common in languages like C and C++, it's always important to keep them in mind. People coding in Python and Java often think they are immune to resource management errors, but that simply isn't true. It's very easy to get to a place where memory exhausts in these languages without realizing that it's happening.

Resource Management Errors: Privoxy

The Privoxy web proxy was allowing remote attackers to cause a denial of service via unspecified vectors.

This failure can be seen in the code snippet in figure 7. There is a block of a piece of a function that is interpreting regular expressions. The problem with the code is that the "l" in the first block may overflow the number of replacements that are allocated, and the error would end up setting the return code to an error and errors would start to gather that would never be freed.

```
/* Backreferences */
if (replacement[i] == '$' && !quoted && i < (int)(length - 1))
{
    char *symbol, symbols[] = "\`+&";
    r->block_length[l] = (size_t)(k - r->block_offset[l]);

    /* Numerical backreferences */
    if (isdigit((int)replacement[i + 1]))
    {
        while (i < (int)length && isdigit((int)replacement[++i]))
        {
            r->backref[l] = r->backref[l] * 10 + replacement[i] - 48;
        }
        if (r->backref[l] > capturecount)
        {
            *errptr = PCRS_WARN_BADREF;
        }
    }
```

Figure 7: Example of the Privoxy failure

How to Fix Privoxy in the Code

Luckily, as seen in figure 8, this fix is fairly straightforward. First, check to ensure the function isn't overflowing the number of replacements. And be sure to free up any resources right away so that you don't gather unnecessary errors in the program.

```
/* Backreferences */
if (replacement[i] == '$' && !quoted && i < (int)(length - 1))
{
    char *symbol, symbols[] = "\`+&";
    if (l >= PCRS_MAX_SUBMATCHES)
    {
        freez(text);
        freez(r);
        *errptr = PCRS_WARN_BADREF;
        return NULL;
    }
    r->block_length[l] = (size_t)(k - r->block_offset[l]);

    /* Numerical backreferences */
    if (isdigit((int)replacement[i + 1]))
    {
        while (i < (int)length && isdigit((int)replacement[++i]))
        {
            r->backref[l] = r->backref[l] * 10 + replacement[i] - 48;
        }
        if (r->backref[l] > capturecount)
        {
            freez(text);
            freez(r);
            *errptr = PCRS_WARN_BADREF;
            return NULL;
        }
    }
```

Figure 8: Example of Privoxy fix

There are a lot of different ways to remediate resource management errors throughout the code development process. Static analysis, for example, will detect a lot of things that may be making it difficult to free errors. Of course, manual analysis can also help by actively examining all the resources, but keep in mind that this is time consuming and difficult, so it's best to use a tool.

And don't forget about design elements. Consider adding C++ wrappers that prevent misused or dangling resources, and examine assumptions on resource limits and balances, and make corrections by simply constructing better code.

Vulnerability 5: Improper Access Control

Improper access control refers to when software does not restrict or incorrectly restricts access to a resource from an unauthorized actor. This includes things like:

- Improper privilege management.
- Improper ownership management.
- Improper authorization.
- Incorrect user management.
- Improper authentication.
- Origin validation error.
- Improper restriction of communication channel to intended endpoints.

Having an improper restriction of communication channels is familiar to many people because it was a part of the Miller/Valasek hack. In that particular case the issue was that the D-Bus service was left open on the cellular data connection. Leaving the D-Bus service open from a cellular channel was a major oversight that would have been caught with a proper security design in place.

Example of Improper Access Control: Stunnel

An example of improper access control, as seen in figure 9, is stunnel (TLS proxy). In this case, this is software that could be used to establish a VPN connection so a secure connection to a backend can exist. In this particular vulnerability, using a redirect option won't actually redirect client connections to the expected server after the initial connection. This will allow for remote attackers to bypass authentication.

```
if(SSL_session_reused(c->ssl)) {
    s_log(LOG_INFO, "SSL %s: previous session reused",
          c->opt->option.client ? "connected" : "accepted");
} else { /* a new session was negotiated */
    new_chain(c);
    if(c->opt->option.client) {
        s_log(LOG_INFO, "SSL connected: new session negotiated");
        enter_critical_section(CRIT_SESSION);
        old_session=c->opt->session;
        c->opt->session=SSL_get1_session(c->ssl); /* store it */
        if(old_session)
            SSL_SESSION_free(old_session); /* release the old one */
        leave_critical_section(CRIT_SESSION);
    } else
        s_log(LOG_INFO, "SSL accepted: new session negotiated");
    print_cipher(c);
}
```

Figure 9: Example of stunnel failure

How Can You Ensure You Have Proper Access Control?

As seen in figure 9, the SSL function is calling a reused session but isn't double-checking the session to make sure that it can be reused. The fix can be seen in figure 10, and demonstrates why it's important to test each session. If a session isn't valid the command can be bailed out of so no unwanted attackers are allowed into a secure connection. This fix makes it possible to verify that everyone logging into the system is in fact allowed to be there. This is a very good example of why short cuts in code should never be used.

```
s_log(LOG_INFO, "SSL %s: %s",
      c->opt->option.client ? "connected" : "accepted",
      SSL_session_reused(c->ssl) ?
      "previous session reused" : "new session negotiated");
if(SSL_session_reused(c->ssl)) {
    c->redirect=(uintptr_t)SSL_SESSION_get_ex_data(SSL_get_session(c->ssl),
                                                   redirect_index);
    if(c->opt->redirect_addr.names && !c->redirect) {
        s_log(LOG_ERR, "No application data found in the reused session");
        longjmp(c->err, 1);
    }
} else { /* a new session was negotiated */
    new_chain(c);
    SSL_SESSION_set_ex_data(SSL_get_session(c->ssl),
                            redirect_index, (void *)c->redirect);
    if(c->opt->option.client) {
        enter_critical_section(CRIT_SESSION);
        old_session=c->opt->session;
        c->opt->session=SSL_get1_session(c->ssl); /* store it */
        if(old_session)
            SSL_SESSION_free(old_session); /* release the old one */
        leave_critical_section(CRIT_SESSION);
    } else { /* SSL server */
        SSL_CTX_add_session(c->opt->ctx, SSL_get_session(c->ssl));
    }
    print_cipher(c);
}
```

Figure 10: Example of stunnel fix

All improper access control problems can be handled by carefully controlling settings, management, and handling privileges, as well as using the principle of least privilege to decide when to drop system privileges. It's also important to compartmentalize a system with safe areas that have unambiguous trust boundaries, to not allow sensitive data to leave trust boundaries, and to exercise caution when interfacing outside of trust boundaries. By doing so it will become extremely challenging for hackers to try and hop around within the system.

Vulnerability 4: Improper Input Validation

Improper input validation includes getting incorrect or missing information from anything that could possibly affect a program's control flow or data flow. This can include things like:

- Improper pathname limitations.
- Improper pathname equivalence resolution.
- External control of configuration settings.
- Improper neutralization (command injection, SQL injection, cross-site scripting, etc.).
- Missing XML validation.
- Improper log neutralization.
- Improper restriction to bounds of memory buffer.
- Improper array index validation.
- Copy into buffer without size check.
- Improper null termination.

Basically anywhere that a developer is getting information from the outside world could be included in this vulnerability.

An Example of Improper Input Validation: SQLite

One example of improper input validation occurs in SQLite, a package that's used in a lot of embedded systems. SQLite does not properly implement comparison operators, which allows context-dependent attackers to cause a denial of service or possibly have unspecified other impacts via a crafted CHECK clause. The example, seen in Figure 11, is a little complicated and maybe a little paranoid, but still something that needs to be taken seriously.

```
}else if( affinity==SQLITE_AFF_TEXT ){
    if( (pIn1->flags & MEM_Str)==0 && (pIn1->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn1->flags & MEM_Int );
        testcase( pIn1->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn1, encoding, 1);
    }
    if( (pIn3->flags & MEM_Str)==0 && (pIn3->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn3->flags & MEM_Int );
        testcase( pIn3->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn3, encoding, 1);
    }
}
    assert( pOp->p4type==P4_COLLSEQ || pOp->p4.pColl==0 );
    if( pIn1->flags & MEM_Zero ){
        sqlite3VdbeMemExpandBlob(pIn1);
        flags1 &= ~MEM_Zero;
    }
    ..........................................................................
}else{
    VdbeBranchTaken(res!=0, (pOp->p5 & SQLITE_NULLEQ)?2:3);
    if( res ){
        pc = pOp->p2-1;
    }
}
/* Undo any changes made by applyAffinity() to the input registers. */

pIn1->flags = flags1;

pIn3->flags = flags3;
break;
```

Figure 11: Example of SQLite failure

The fail in figure 11 is that a couple of flags are assigned that end up undoing actions taken by the routine. The overall problem here is that the wrong path through the code can end up resetting those flags and making the memory appear as if it is dynamically allocated when it wasn't, or vice versa.

This is obviously bad because an attacker could use this vulnerability to get information that they aren't supposed to have access to, or cause memory to free that wasn't supposed to be freed.

How to Fix the SQLite Vulnera

The fix for this vulnerability, as seen in figure 12, is a little bit complicated, but basically what needs to be done is that the areas that have the flag need to save off the the important bits to direct memory every time. This will ensure that when developers do start playing around with multiple flag values that they aren't allowed to reset specific bits.

```
}else if( affinity==SQLITE_AFF_TEXT ){
    if( (pIn1->flags & MEM_Str)==0 && (pIn1->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn1->flags & MEM_Int );
        testcase( pIn1->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn1, encoding, 1);
        testcase( (flags1&MEM_Dyn) != (pIn1->flags&MEM_Dyn) );
        flags1 = (pIn1->flags & ~MEM_TypeMask) | (flags1 & MEM_TypeMask);
    }
    if( (pIn3->flags & MEM_Str)==0 && (pIn3->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn3->flags & MEM_Int );
        testcase( pIn3->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn3, encoding, 1);
        testcase( (flags3&MEM_Dyn) != (pIn3->flags&MEM_Dyn) );
        flags3 = (pIn3->flags & ~MEM_TypeMask) | (flags3 & MEM_TypeMask);
    }
}
    assert( pOp->p4type==P4_COLLSEQ || pOp->p4.pColl==0 );
    if( pIn1->flags & MEM_Zero ){
        sqlite3VdbeMemExpandBlob(pIn1);
        flags1 &= ~MEM_Zero;
    }
    ..........................................................................
}else{
    VdbeBranchTaken(res!=0, (pOp->p5 & SQLITE_NULLEQ)?2:3);
    if( res ){
        pc = pOp->p2-1;
    }
}
/* Undo any changes made by applyAffinity() to the input registers. */
assert( (pIn1->flags & MEM_Dyn) == (flags1 & MEM_Dyn) );
pIn1->flags = flags1;
assert( (pIn3->flags & MEM_Dyn) == (flags3 & MEM_Dyn) );
pIn3->flags = flags3;
break;
```

Figure 12: Example of SQLite fix

However, the flag in figure 11 is actually really prone to usage errors and could use a redesign to ensure that it can perform safely.

Often improper input validation errors can be fixed through correct architecture and design, proper implementation, and by utilizing automated static analysis.

When designing code be sure to check data on both client and server sides of transaction – don't assume that the server side is only passing along intended information, actually make sure that this is the case.

SCA tools are very good at finding unclean, unquoted, or unescaped data, so be sure to use a SCA tool when designing code.

Vulnerability 3: Information Exposure

Information exposure is intentional or unintentional disclosure of information to an actor that isn't explicitly authorized. This information exposure tends to happen through:

- Sent data.
- Data queries.
- Discrepancy.
- Error messages.
- Debug messages.
- Process environment.
- Caching.
- Indexing private data.

Developers tend to forget about information exposure through error messages and debug messages, as well as any information that may find its way into a log file. For that reason, developers need to be mindful of everything that lives in the log file, and be confident that there isn't any information sitting there that may be of value to an attacker.

Example of Information Exposure: Adobe AIR

Adobe AIR is an interesting example of why you shouldn't just give information away. Adobe Flash Player wasn't properly restricting discovery of memory addresses, which would allow an attacker to bypass the address space layout randomization (ASLR) protection mechanism through unspecified vectors. This makes the process of saving memory to random locations a waste because attackers are able to find all the memory.

Though this may seem like a little issue, it's important to remember that most attackers need more than one vulnerability to make an impact. That is why developers need really need to be careful about what memory is released, so attackers aren't able to use that information in addition with other information or another vulnerability.

How to Fix Information Exposures

By compartmentalizing systems with safe areas, by not allowing sensitive data to leave trust boundaries, and by exercising caution when interfacing outside of these trust boundaries, developers can securely design code to protect from these information exposures.

Code analysis should also be used to secure information. Automated SCA can perform context-dependent weakness analysis, dynamic code analysis can perform fuzz testing and allows testing within a monitored virtual environment. Manual code analysis is still important to ensure that there are no areas of external contact for unintended data access and that there is no information shared unless it is absolutely necessary. In addition, this also includes error and debug messages.

Vulnerability 2: Access Control

Access control is any weakness related to the management of permissions, privileges, or other security features. This includes:

- Sandbox issues (chroot environments).
- Permission issues (Improper inheritance, permissive defaults, symbolic links subverting permissions, improper permission preservation, etc.).
- Improper ownership management.
- Improper access control.

Example of Access Control: Kerberos

As seen in figure 13, Kerberos is not properly tracking whether a client's request has been validated. This is a common protocol to use for client and servers to authenticate each other, and can be a place where vulnerabilities live.

```
}else if( affinity==SQLITE_AFF_TEXT ){
    if( (pIn1->flags & MEM_Str)==0 && (pIn1->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn1->flags & MEM_Int );
        testcase( pIn1->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn1, encoding, 1);
        testcase( (flags1&MEM_Dyn) != (pIn1->flags&MEM_Dyn) );
        flags1 = (pIn1->flags & ~MEM_TypeMask) | (flags1 & MEM_TypeMask);
    }
    if( (pIn3->flags & MEM_Str)==0 && (pIn3->flags & (MEM_Int|MEM_Real))!=0 ){
        testcase( pIn3->flags & MEM_Int );
        testcase( pIn3->flags & MEM_Real );
        sqlite3VdbeMemStringify(pIn3, encoding, 1);
        testcase( (flags3&MEM_Dyn) != (pIn3->flags&MEM_Dyn) );
        flags3 = (pIn3->flags & ~MEM_TypeMask) | (flags3 & MEM_TypeMask);
    }
}
    assert( pOp->p4type==P4_COLLSEQ || pOp->p4.pColl==0 );
    if( pIn1->flags & MEM_Zero ){
        sqlite3VdbeMemExpandBlob(pIn1);
        flags1 &= ~MEM_Zero;
    }
    ..........................................................................
}else{
    VdbeBranchTaken(res!=0, (pOp->p5 & SQLITE_NULLEQ)?2:3);
    if( res ){
        pc = pOp->p2-1;
    }
}
/* Undo any changes made by applyAffinity() to the input registers. */
assert( (pIn1->flags & MEM_Dyn) == (flags1 & MEM_Dyn) );
pIn1->flags = flags1;
assert( (pIn3->flags & MEM_Dyn) == (flags3 & MEM_Dyn) );
pIn3->flags = flags3;
break;
```

Figure 13: Example of Kerberos failure

How to Fix the Kerberos Issue

Figure 14 demonstrates that in order to fix this issue developers shouldn't assume the default state is preauthorized. By not authorizing the code early, errors will be caught before the code is claimed as authorized. This issue is a simple problem to have — but it is also easy to fix.

```
static void
on_response(void *data, krb5_error_code retval, otp_response response)
{
    struct request_state rs = *(struct request_state *)data;

    free(data);

    if (retval == 0 && response != otp_response_success)
        retval = KRB5_PREAUTH_FAILED;

    if (retval == 0)
        rs.enc_tkt_reply->flags |= TKT_FLG_PRE_AUTH;

    rs.respond(rs.arg, retval, NULL, NULL, NULL);
}


static void
otp_verify(krb5_context context, krb5_data *req_pkt, krb5_kdc_req *request,
           krb5_enc_tkt_part *enc_tkt_reply, krb5_pa_data *pa,
           krb5_kdcpreauth_callbacks cb, krb5_kdcpreauth_rock rock,
           krb5_kdcpreauth_moddata moddata,
           krb5_kdcpreauth_verify_respond_fn respond, void *arg)
{
    krb5_keyblock *armor_key = NULL;
    krb5_pa_otp_req *req = NULL;
    struct request_state *rs;
    krb5_error_code retval;
    krb5_data d, plaintext;
    char *config;

    //DELETED: enc_tkt_reply->flags |= TKT_FLG_PRE_AUTH;
```

Figure 14: Example of Kerberos fix

In order to stay in charge of access control vulnerabilities, it's important to keep architecture and design top of mind. It is always important to correct by construction, and to make sure that code is functioning optimally from the beginning. Also remember to use manual code analysis and carefully examine all access control granting and handoff for assumptions, defaults, and error paths.

Don't forget to use other tools on your code — like static and dynamic analysis. Fuzz testing and context dependent weakness analysis could save a major headache.

Vulnerability 1: Memory Buffer Problems

Memory buffer problems occur when software can read or write to locations outside of the boundaries of the memory buffer.

- This includes:
- Not checking size of input on copy.
- Bug allowing writing to arbitrary locations.
- Out-of-bounds read.
- Pointers outside expected range.
- Untrusted pointer dereference.
- Uninitialized pointers.
- Expired pointer references.
- Access of memory beyond buffer end.

Example of Memory Buffer Problems: GNU libc

Figure 15 shows an example of memory buffer problems in GNU libc. This example may come as a surprise, but even highly-tested, highly-utilized, highly-examined libraries can have security bugs. This shows that no developer or program is immune to mistakes or code issues.

```
if (__glibc_unlikely (wpsize == wpmax))                                 \
 {                                                                      \
   CHAR_T *old = wp;                                                    \
   size_t newsize = (UCHAR_MAX + 1 > 2 * wpmax                          \
                     ? UCHAR_MAX + 1 : 2 * wpmax);                      \
   if (use_malloc || !__libc_use_alloca (newsize))                      \
     {                                                                  \
       wp = realloc (use_malloc ? wp : NULL, newsize);                  \
       if (wp == NULL)                                                  \
         {                                                              \
           if (use_malloc)                                              \
             free (old);                                                \
           done = EOF;                                                  \
           goto errout;                                                 \
         }                                                              \
       if (! use_malloc)                                                \
         MEMCPY (wp, old, wpsize);                                      \
       wpmax = newsize;                                                 \
       use_malloc = true;                                               \
     }                                                                  \
   else                                                                 \
     {                                                                  \
       size_t s = wpmax * sizeof (CHAR_T);                              \
       wp = (CHAR_T *) extend_alloca (wp, s,                            \
                                      newsize * sizeof (CHAR_T));       \
       wpmax = s / sizeof (CHAR_T);                                     \
       if (old != NULL)                                                 \
         MEMCPY (wp, old, wpsize);                                      \
```

Figure 15: Example of GNU libc failure

This particular example is wscanf allocating too little memory in particular conditions. As seen in figure 15, it really is all about whether or not developers account for wide characters.

How to Fix GNU Libc

The fix, as displayed in figure 16, is basically just inserting a couple sizeof {CHAR_T} functions into the code. So instead of allocating for the number of bytes, developers should allocate for size of international characters.

```
if (__glibc_unlikely (wpsize == wpmax))                          \
 {                                                               \
   CHAR_T *old = wp;                                             \
   bool fits = __glibc_likely (wpmax <= SIZE_MAX / sizeof (CHAR_T) / 2); \
   size_t wpneed = MAX (UCHAR_MAX + 1, 2 * wpmax);               \
   size_t newsize = fits ? wpneed * sizeof (CHAR_T) : SIZE_MAX;  \
   if (!__libc_use_alloca (newsize))                             \
     {                                                           \
       wp = realloc (use_malloc ? wp : NULL, newsize);           \
       if (wp == NULL)                                           \
         {                                                       \
           if (use_malloc)                                       \
             free (old);                                         \
           done = EOF;                                           \
           goto errout;                                          \
         }                                                       \
       if (! use_malloc)                                         \
         MEMCPY (wp, old, wpsize);                               \
       wpmax = wpneed;                                           \
       use_malloc = true;                                        \
     }                                                           \
   else                                                          \
     {                                                           \
       size_t s = wpmax * sizeof (CHAR_T);                       \
       wp = (CHAR_T *) extend_alloca (wp, s, newsize);           \
       wpmax = s / sizeof (CHAR_T);                              \
       if (old != NULL)                                          \
         MEMCPY (wp, old, wpsize);                               \
```

Figure 16: Example of GNU libc fix

How to Stay Safe Among Hackers

By staying aware of what are the top 10 vulnerabilities, you can ensure that your code remains secure. This means that staying aware and secure for the top 10 vulnerabilities will put code in a much better place. Of course, it's always important to stay current on software vulnerabilities and patches — new issues are being found very regularly.

4 Best Practices for Protecting Against Cybersecurity Vulnerabilities

There are four best practices to ensure protection from common vulnerabilities: clean design, methodical process, careful analysis, and good tools.

Clean Design: Make a design that cleanly separates processes with different security needs, and that has a tightly controlled interface between components. Don't provide more access to system components than is strictly necessary. And don't reinvent the security wheel — use tried and true technologies to authenticate, encrypt, and secure your product. If there are any doubts about the security of a design, don't be afraid to ask for assistance from a security professional.

Methodical Process: Processes only work when they're being followed. Don't let process take all the joy ut of programming, but let it help create a safety net for software. Knowing that processes can catch silly mistakes doesn't make a developer careless — it makes them confident.

Careful Analysis: Make sure to carefully think about every place a hacker can approach the system. Don't be trapped by conventional thinking about the existing interfaces being designed. Carefully examine each interface and make sure that they're as resilient as possible. Also examine where a program is consuming data, don't take anything for granted.

Good Tools: Many of the errors found by static code analysis will translate directly into closing vulnerabilities. Using tools to clean up sloppy coding practices will result in tighter, and more secure code. And many tools have special configurations designed to look for specific security problems.

The Right Tools for the Right Processes

Build: Integrate tools as part of the build process from the very beginning to take advantage of consistent checks in incremental daily builds, ensuring that the insertion of tools into a mature build process is not quite such a daunting task. Don't wait until near the end of the project when high inertia and bug introduction risk will lead to opting out of many little fixes that the tools may indicate.

Validation: Validation depends on input from not only internal code, but code that is relied on as well. Find tools that can help flow integrity checks throughout the entire integration process. And don't stop running tools just because a program is in production. Rely on them to double-check bug fixes or feature additions after the golden build to make sure that any new code added adheres to the same high quality bar set initially.

Review Black Boxes

Things like font engines, web browsers, PDF viewers, speech recognition engines, graphics toolkits, and Adobe Flash and AIR all need to be considered when developing. Just because that code wasn't created internally doesn't mean that it can't be hacked — look for ways to validate any inputs those modules receive or to isolate those components as much as possible from the rest of the system, so that any inadvertent security breach cannot propagate far.

Perform Regular Updates

Paraphrasing a famous security quote: a software product is never 100 % secure from security risk, unless it's disconnected from every possible input and encased in a block of concrete, and even then it's doubtful. It is a maximum of the security world that security updates are a necessary evil because software is never perfect. Embedded software are no different — without having a way to issue security patches, the game is already lost. Build software processes assuming that production does not stop software development, and that the system will need to have the technology and infrastructure in place to receive updates.

Awareness of these top 10 issues can help with nearly 90% of all vulnerabilities in embedded software. Other 10% by relying on the right tools, good design, well thought-out processes, and careful analysis.

Why Choose Perforce Static Analyzers to Help Prevent Cybersecurity Vulnerabilities

Perforce's static code analyzers have been trusted for over 30 years to deliver the most accurate and precise results to mission-critical project teams across a variety of industries.

Klocwork is the most accurate code analyzer supporting C, C++, C#, and Java programming languages. It scales to projects of any size and is very effective within a continuous integration (CI) and DevOps environment. What's more, it's Differential Analysis provides developers with the shortest possible analysis times.

By using Klocwork, you can be reassured that your code will be secure, reliable, and compliant.

See for yourself the impact that Klocwork can have on the quality of your code by starting your free trial.

Start Trial

perforce.com/products/sca/free-static-code-analyzer-trial

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle.

Our portfolio includes solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static & dynamic code analysis, version control, and more. With over 9,000 customers, Perforce is trusted by the world's leading brands, including NVIDIA, Pixar, Scania, Ubisoft, and VMware. For more information, visit perforce.com.

Contact Us

perforce.com/support