



Developed by a team of researchers in the Computer Systems Lab at Stanford University, SWAT's source code analysis technology is based on powerful patent pending statistical learning techniques combined with a comprehensive abstract dataflow engine. Initial applications resulted in the successful detection of over 2000 defects in Linux, including hundreds of security holes.

This whitepaper illustrates a number of real bugs that SWAT identified in the Linux kernel. Although SWAT finds many other classes of defects (see <http://www.coverity.com/coverity-checkerlist.pdf>), the following examples illustrate SWAT's basic capabilities.

Illegal Pointer Accesses

NULL pointer dereferences are a common type of software defect that result in system crashes. In this Linux example, SWAT determines that `create_proc_entry()` can potentially return a NULL pointer in line 366. Pointer `p`, however, is not checked to be a valid non-NULL pointer before it is dereferenced in line 367. If `p` indeed was NULL, the system will crash.

When there is an assignment or a function call that can set a pointer to NULL, SWAT checks every path through the code to ensure that the pointer is checked to be valid before it is accessed. SWAT uses sophisticated interprocedural analyses and statistical learning techniques to assess which assignments or function calls should be targeted. Further, SWAT uses powerful data-flow analyses to track each pointer, even if it is reassigned to other variables or fields.

```
==>
NULL detector: pointer p is potentially NULL
File: /linux/2.4.4/drivers/media/video/videodev.c
Function: videodev_proc_create_dev
Line: 366
    p = create_proc_entry(name, S_IFREG|S_IRUGO|S_IWUSR, video_dev_proc_entry);

==>
NULL detector: Illegally dereferencing p
Line: 367
    p->data = vfd;
    p->read_proc = videodev_proc_read;
    d->proc_entry = p;
```

Memory Leaks

Memory leaks are infamous for being pervasive and difficult to diagnose. Although runtime tools such as Purify can help locate some of the leaks, they only check parts of the code that are actually executed during testing. For large code bases, runtime tools result in monitoring only a very small fraction of the possible execution paths. Further, runtime tools do not diagnose the root cause of the leak.

On the other hand, SWAT detects memory leaks at compile time and checks all possible paths through the code. In the example below, SWAT automatically infers that `alloc_skb()` is a memory allocation function and discovers a scenario where the enclosing routine exits without freeing the allocated memory (`skb`), resulting in a leak.

```
==>
LEAK detector: allocating storage skb
File: /linux/2.5.48/drivers/isdn/tpam/tpam_queues.c
Function: tpam_irq
Line: 112
    if (!(skb = alloc_skb(sizeof(skb_header) + sizeof(pci_mpb) +
    ...
    do {
        hpic = readl(card->bar0 + TPAM_HPIC_REGISTER);
        if (waiting_too_long++ > 0xffffffff) {
            spin_unlock(&card->lock);
            printk(KERN_ERR "TurboPAM(tpam_irq):" "waiting too long...\n");
        }
    } while (hpic & 0x00000002);
    ...
    kfree_skb(skb);
```

Use After Free

SWAT can also detect use-after-free bugs, where memory is accessed after it is freed. In the following example, the programmer erroneously uses index `i` instead of `j` in line 822, freeing the same memory location `ipp_table[i]` multiple times. Once a pointer is freed, SWAT checks all ensuing paths to see if it is accessed (along iterations around the inner for-loop in this example).

This particular error is precisely the type that would be difficult to catch without SWAT since the actual error would not be hit at runtime unless a hard-to-test condition in the if-statement is true.

```

for (i = 0; i < ISDN_MAX_CHANNELS; i++) {
    if (!(ipp_p_table[i] = (struct ipp_p_struct *) kmalloc(sizeof(struct ipp_p_struct),
                                                         GFP_KERNEL))) {
        printk(KERN_WARNING " isdn_ipp_init: Could not alloc ipp_p_table\n");
        for (j = 0; j < i; j++)

```

==>

USE-AFTER-FREE detector: using freed storage ipp_p_table[!]

File: /linux/2.4.4-ac8/drivers/isdn/isdn_ipp.c

Function: isdn_ipp_init

Line: 822

```

        kfree(ipp_p_table[i]);

```

Buffer Overrun

Unlike other defect-detection tools, SWAT is not limited to finding only NULL dereferences or simple memory leaks. SWAT's sophisticated analysis engine can also locate cases of buffer overruns, in which data is modified outside of the logical bounds of arrays or buffers. Buffer overruns are particularly difficult to diagnose since the effects of the violations commonly manifest millions of cycles after the errors occur, making it difficult to trace back to the root causes.

SWAT detected many cases of buffer overruns in Linux, including the one shown below. The character array **buf** is declared with size 20. Pointer **base** addresses the fourth element in **buf**. On line 1170, however, **base[16]** which is the 17th element of **base** and the 21st element of **buf** is accessed, modifying data that is outside the bounds of the 20 character buffer. This may result in the corruption of an unrelated part of memory with detrimental effects. Buffer overruns are also frequently sources of security vulnerabilities.

```

unsigned char buf[20], *base;
struct dvd_layer *layer;
...
base = &buf[4];
layer = &s->physical.layer[layer_num];
...
layer->linear_density = base[3] >> 4;
layer->start_sector = base[5] << 16 | base[6] << 8 | base[7];
layer->end_sector = base[9] << 16 | base[10] << 8 | base[11];
layer->end_sector_I0 = base[13] << 16 | base[14] << 8 | base[15];

```

==>

```
BUFFER OVERRUN detector: index 16 is out of bounds
```

```
FILE: /linux/2.5.48/drivers/cdrom/cdrom.c
```

```
Function: dvd_read_physical
```

```
Line: 1170
```

```
    layer->bca = base[16] >> 7;  
    return 0;  
}
```

Concurrency Errors

A unique feature of SWAT is its ability to detect concurrency bugs. Deadlocks and race conditions are some of the most difficult problems to debug. Error scenarios are nearly impossible to reproduce, since they rely on a particular scheduling of threads. Concurrency problems can manifest in systems “hanging” or results being unpredictable. SWAT is the only tool that can automatically detect concurrency errors in non-trivial software programs.

SWAT detected the following deadlock in Linux. Thread 1 acquires lock **im->lock** and eventually tries to acquire lock **inetdev_lock** in devinet.c:759. Thread 2 in the meantime acquires lock **inetdev_lock**, which Thread 1 will need, and eventually tries to acquire lock **in_dev->lock**. Finally, Thread 3 acquires lock **in_dev->lock**, which Thread 2 will need, and, in turn, tries to acquire lock **im->lock**. If scheduled in a certain way, Thread 1 will wait on a lock that Thread 2 owns, Thread 2 will wait on a lock that Thread 3 owns, and, finally, Thread 3 will wait on a lock that Thread 1 owns. This circularity in the hold-wait relationships among the different locks leads to a situation where none of the threads can progress, resulting in a deadlock.

```
==>
```

```
DEADLOCK detector: lock cycle detected
```

```
Thread 1: im->lock --> inetdev_lock
```

```
File: /linux/net/ipv4/igmp.c
```

```
Line: 268
```

```
static void igmp_timer_expire(unsigned long data)  
{  
    ...  
    spin_lock(&im->lock);  
    im->tm_running=0;  
  
    if (IGMP_V1_SEEN(in_dev))  
        err = igmp_send_report(...);  
        -> ip_route_output_key:2149  
        -> __ip_route_output_key:2142  
        -> ip_route_output_slow:1988  
        -> fib_semantics.c:__fib_res_prefsrc:638  
        -> devinet.c:inet_select_addr:759  
        read_lock(&inetdev_lock);
```

==>

Thread 2: inetdev_lock --> in_dev->lock

File: /linux/net/ipv4/devinet.c

Line: 759

```
u32 inet_select_addr(const struct net_device *dv, u32 dst, int scope)
{
    u32 addr = 0;
    struct in_device *in_dev;

    read_lock(&inetdev_lock);
    in_dev = __in_dev_get(dev);
    if (!in_dev)
        goto out_unlock_inetdev;
    read_lock(&in_dev->lock);
```

==>

Thread 3: in_dev->lock --> im->lock

File: /linux/net/ipv4/igmp.c

Line: 338

```
static void igmp_heard_query(struct in_device *in_dev, ...)
{
    ...
    read_lock(&in_dev->lock);
    for(im=in_dev->mc_list; im!=NULL; im=im->next) {
        ...
        igmp_mod_timer(im, max_delay);
        -> igmp_mod_timer:165
        spin_lock_bh(&im->lock);
```

Security Vulnerabilities

Another powerful feature of SWAT is its ability to detect security vulnerabilities. SWAT is the only commercially available tool that can pinpoint security holes at the source code level. Malicious hackers can exploit security holes in the program to gain control of the system, corrupt or compromise data, or significantly degrade performance (DOS attacks).

In the following example, SWAT detected the use of tainted data that was used in a memory copying routine without first performing sanitization. The structure **d** is copied from an untrusted processes into kernel memory in line 1411 using the routine **copy_from_user()**. The field **d.idx** is checked for overflow and underflow immediately, but the remaining fields remain tainted. On line 1419, two of these tainted fields, **d.address** and **d.used** are passed to another call to **copy_from_user()**, allowing the user to overflow the kernel buffer with memory from an arbitrary address. Variants of this sort of attack are commonly used by hackers to attain root privileges.

```

==>
SECURITY HOLE detector: Data d tainted
File: /linux/2.4.6/drivers/char/drm/i810_dma.c
Function: i810_copybuf
Line: 1411
    if (copy_from_user(&d, (drm_i810_copy_t *)arg, sizeof(d)))
        return -EFAULT;

    if(d.idx < 0 || d.idx > dma->buf_count) return -EINVAL;
    buf = dma->buflist[ d.idx ];
    buf_priv = buf->dev_private;
    if (buf_priv->currently_mapped != I810_BUF_MAPPED) return -EPERM;

==>
SECURITY HOLE detector: Use of tainted data 'd.address'
SECURITY HOLE detector: Use of tainted data 'd.used'
Line: 1419
    if (copy_from_user(buf_priv->virtual, d.address, d.used))
        return -EFAULT;

    sarea_priv->last_dispatch = (int) hw_status[5];

```

Statistical Inference

SWAT uses patent-pending statistical learning techniques to adapt to each new code base, accounting for company-specific conventions and coding styles. In addition, SWAT can infer domain-specific rules to check without knowing what they are a priori. The Linux bug illustrated below is one example.

SWAT tracks all pairs of events in the code, such as pairs of function calls, and detects cases where the programmer significantly deviates from the rest of the code base. In this example, SWAT infers that a call to **lock_kernel()** is generally followed by a call to **unlock_kernel()**. When it finds a deviant case along a path where **unlock_kernel()** is not called (because of a return in line 2058), SWAT flags it as a potential error. Using the same statistical learning techniques, SWAT also ranks such errors based on how likely they are to be real bugs.

```

==>
STATISTICAL A-B detector: lock_kernel is not followed by unlock_kernel
File: drivers/sound/cmpci.c
Function : cm_midi_release
Line: 2058
    lock_kernel();
    if (file->f_mode & FMODE_WRITE) {

```

```
add_wait_queue(&s->midi.owait, &wait);
for (;;) {
    ...
    if (count <= 0)
        break;
    if (signal_pending(current))
        break;
    if (file->f_flags & O_NONBLOCK) {
        remove_wait_queue(&s->midi.owait, &wait);
        set_current_state(TASK_RUNNING);
    }
}
```

==>

STATISTICAL A-B detector: Missing call to unlock_kernel

Line: 2073

```
    return -EBUSY;
}
...
unlock_kernel();
```

Summary

The bugs described in this whitepaper illustrated actual Linux bugs that were fixed as a result of SWAT's analysis. Although they give a sense for the general capabilities of SWAT, these errors only show the tip of the iceberg. SWAT detected many other types of errors among the thousands of Linux bugs that it found, and has since been applied to major code bases at some of the largest software companies in the world with equal success.

The analyses that uncovered the Linux bugs shown above are generic checks that come prepackaged with SWAT. Unlike other tools, SWAT can also be easily extended with company-specific checks using simple to use script and API level interfaces. Company coding policies and API rules can be turned into SWAT checks, and bugs found by QA after they are hit (in the field) can also be translated into SWAT analyses to detect similar errors in other parts of the code.

About Coverity

Coverity, Inc. is a leading provider of source code analysis solutions that help organizations produce reliable, secure software while significantly improving time to market. Coverity's quickly growing customer base includes a wide range of companies, from startups to Fortune 100 enterprises. Coverity, Inc. is headquartered in Menlo Park, California. For more information or a free trial, contact us at:

Tel: 1-650-980-3408

E-mail: info@coverity.com