Using Programmer-Written Compiler Extensions to Catch Security Holes

Ken Ashcraft and Dawson Engler*
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

Abstract

This paper shows how system-specific static analysis can find security errors that violate rules such as "integers from untrusted sources must be sanitized before use" and "do not dereference user-supplied pointers." In our approach, programmers write system-specific extensions that are linked into the compiler and check their code for errors. We demonstrate the approach's effectiveness by using it to find over 100 security errors in Linux and OpenBSD, over 50 of which have led to kernel patches. An unusual feature of our approach is the use of methods to automatically detect when we miss code actions that should be checked.

1 Introduction

Secure code must obey rules such as "sanitize untrusted input before using it," "check permissions before doing operation X," "do not release sensitive data to unauthorized users." All code must obey these rules and a single violation can compromise the integrity of the entire system. Unfortunately, many rules are poorly understood and erratically obeyed.

This paper shows how to combat these problems by automatically checking security rules in a general, lightweight manner using system-specific, static analysis. We have used our approach, *metacompilation* (MC), in prior work to find hundreds of errors in operating system (OS) code [8, 9]. The current paper focuses on using MC to check security properties. Our results show it catches security errors as effectively as it did non-security ones.

*This research was supported in part by DARPA contract MDA904-98-C-A933, by SAL contract NAS1-98139, and by a grant from the Stanford Networking Research Center.

MC finds rule errors by exploiting the fact that many of the abstract properties relevant to a rule map clearly to concrete code actions [8]. Thus, we can check these actions for errors using a compiler. For example, to catch violations of the rule "do not use an unchecked, untrusted value as an array index" a compiler can follow integers read from untrustworthy sources and flag when they are not bounds checked before being used as an index.

Of course, to check a rule, the compiler must first know it. Since many security rules are domain or even system specific, hard wiring a fixed set into the compiler is ineffective. MC attacks this problem by making it easy for implementers to add rules to the compiler in the form of high-level, system-specific checkers. These extensions encode the rule to be checked, but leave the bulk of the analysis to the compiler. Because checkers are written by system implementers they can readily take into account the ad hoc (sometimes bizarre) semantics of a system.

When applicable, the approach has several nice features. First, it propagates the knowledge of one programmer to many. Rather than requiring all programmers to know and obey all of the rules all of the time, MC allows a single programmer to understand a rule once, write an extension that checks it, and then have this extension automatically imposed on all code. This ability is important for security, since many security rules are subtle. For several of the errors we found, even seasoned implementers did not understand the relevant rule, to the extent that they initially argued against the validity of the errors we reported (until other implementers explained the attacks).

Second, static analysis readily finds difficult-to-observe errors. This ability is important since many security errors are silent: they compromise system security for certain input values but may not crash the machine. In contrast, static analysis can say what line and what file has an error and why.

Third, static analysis catches errors without running code. This feature helps to find errors in operating systems, which have too many code paths for thorough testing. Worse, most operating system code cannot be executed: most of it resides in device drivers, and a given site typically tens of devices rather than the 100-1000s needed to test all drivers. Many of the driver errors we found were for devices we have never heard of, much less have access to.

Finally, extensions are lightweight. Once the fixed cost of writing the extension is paid, there is little incremental manual cost to using it as code size grows (mainly the cost of inspecting errors). Other approaches, such as formal verification and testing, require work proportional to the size of code.

We illustrate how to use metacompilation to find security bugs by examining a single checker deeply and sketching two others. Our main focus is our range checker, which warns when integers read from untrusted sources (such as network packets) are used in dangerous ways without first being checked. This checker both demonstrates the approach's effectiveness and serves as a case study illustrating the general issues involved in building useful checkers.

Our two other checkers show the generality of the approach. The first warns when pointers read from untrusted sources are dereferenced. The second warns when non-security errors in the kernel could be induced by the user, thus letting a malicious attacker crash the system.

A key result of the paper is showing how to make checkers "fail stop" by catching when they miss actions relevant to the rule they check. We do so using *belief analysis* [9]. Belief analysis infers programmer beliefs from source code. We use it to find when an extension writer has missed a checkable action by flagging when code appears to believe an action should be checked, even though the extension does not. For example, in the context of the range checker, if code reads a value from a dangerous location and then sanitizes it, this implies the programmer believes the value will be used for something dangerous. If this value does not reach a known dangerous operation, this implies that we have missed one. Other checking approaches can also use this technique, since they are all vulnerable to incomplete or wrong specifications.

Our most important result is that MC checkers are effective on real code. The range checker alone found well over one hundred security errors in Linux and OpenBSD code, almost half of which have resulted in kernel patches based on our reports.

A final practical result is that writing checkers requires little background knowledge. The bulk of the implementation work for this paper was done by the first author as an undergraduate with little compiler training and no real prior exposure to OS code.

```
/* 2.4.5-ac8/drivers/usb/se401.c:se401_ioctl */
if (copy_from_user(&frame, arg, sizeof(int)))
    return -EFAULT;
ret=se401_newframe(se401, frame);
se401->frame[frame].grabstate=FRAME_UNUSED;
return ret;
```

Figure 1: Simple range error that can compromise integrity by corrupting kernel memory. The variable `frame` is read from the user and then used as an unchecked array index to the `se401`→`frame` array.

While the approach works well on real code, there are three main caveats to keep in mind. First, our checkers are not verifiers: code that contains errors can pass through them silently. Second and conversely, the checkers give errors when they should not. False positives can arise both because the checker's analysis is overly simplistic or because static analysis is undecidable in general. Despite this, the false positive rate is quite low — less than 20% in most cases. Finally, since we are not Linux or BSD implementers, we are capable of mis-diagnosing error reports. We have countered this problem by releasing all of our errors to the system builders for confirmation.

Sections 2— 6 describes our integer range checker in detail. Sections 2 gives an overview; Section 3 shows how we infer missed actions; Section 4 explores analysis issues; Section 5 gives examples of enforcing subtle rules; and Section 6 presents our results. Section 7 explains both the checking of user-supplied pointers and marking user-exploitable errors. Section 8 discusses related work and Section 9 concludes.

2 Range Checker Overview

This section gives an overview of our range checker, which warns when integers supplied by untrustworthy sources are used for dangerous operations before performing necessary range checks.

Figure 1 gives an example error caught by the checker. The variable `frame` is read from a user application (via `copy_from_user`) and then used as an unchecked array index. Additionally, because the checker does inter-procedural analysis (see § 4) it warns that `frame` is passed to the function `se401_newframe`, which similarly uses it as an index. Both errors let an attacker corrupt arbitrary kernel memory.

We give an overview of the checker below; the next three sections discuss several aspects in more detail.

Abstractly, the checker ensures that data pro-

Source	Description
sys_*(T x)	All system call parameters of any type. In both BSD and Linux system calls have a "sys_" prefix.
skb→data	Data read from a network packet. On Linux, network data is initially read from the data field of an skb structure. We do not do this check for BSD.
copyin(u,k,sz)	Data explicitly copied from user address u into the memory pointed to by k. Additionally, the variable u is also tainted since user addresses are almost certainly supplied by the user (see § 3.1 for details).

Table 1: The three sources of untrustworthy data.

```
/* 2.4.5/drivers/char/drm/i810_dma.c:1417:
     i810_copybuf */
if(copy_from_user(&d, arg, sizeof(d)))
    return -EFAULT;
if(d.idx > dma->buf_count)
    return -EINVAL;
buf = dma->buflist[ d.idx ];
...
if (copy_from_user(buf_priv->virtual,
                     d.address, d.used))
    return -EFAULT;
```

Figure 2: Missing lower bound check that allows a read of arbitrary kernel memory. The structure `d` is read from the user. Its signed integer field `d.idx` is upper-bound checked but not lower-bound checked before being used as an array index. As a bonus, the field `d.used` is completely unchecked, allowing up to 2GB of user data to be copied into the kernel from user space.

duced by an untrustworthy source does not reach a trusting sink without being sanitized. Thus, it is specified by three pieces: (1) the untrustworthy sources that generate data, (2) the checks that must be done to sanitize the data from these sources, and (3) the trusting sinks that must be protected. We discuss these below.

Untrustworthy sources. The checker tracks untrusted data from three sources (summarized in Table 1):

1. System calls. These typically adhere to well-defined naming conventions (e.g., on both Linux and BSD their names begin with the prefix "`sys_`"). To detect these routines, the checker calls a user-supplied function that returns 1 if a given function name could be a system call, 0 if it could not.

2. The routines that can copy data from user space. These routines are supplied as a text file giving the routine name and the argument that represents the copied in data. For example, on Linux the routines `copy_from_user` and `get_user` copy data into their first argument. On BSD `copyin` copies user data into the second. Typically, there are between two to ten such routines.

3. Data from the network. This requires specifying which data structures contain packet data and how it is extracted as well as which routines send and receive data. We currently only support this check on Linux. Section 3.3 gives more detail.

Sanitizing checks. In general, signed integers must be both upper- and lower-bounds checked; unsigned integers need only an upper-bound check. Untrusted integers can also be sanitized with an equality check, since this bounds the integer to single number (the degenerate range). While the need for these checks is simple to understand, the possibility of integer overflow can make them difficult to obey in practice (see § 5). Many programmers appear to view integers as having arbitrary precision, rather than being fixed-sized quantities operated on with modulo arithmetic.

A common error is to store a user-supplied length or index in a signed variable but then only perform an upper-bound check on it, allowing an attacker to compromise the system by passing in negative values. Figure 2 gives a representative example. Here the data in structure `d` is copied from user space. The signed field `d.idx` is checked against an upper bound, but not against a lower bound. Thus, an attacker could pass in a negative value to force the kernel to read from arbitrary memory.

It is important to note that the checker is explicitly unsound in that it ensures that an untrusted value is checked against *some* bound rather than against the correct one. The lack of any specification of sizes in real code makes precise, static bounds checking challenging. Our decision to not model accurate bounds means we can use simple analysis to

Sinks	Description
`array[x]`	Array index: corrupt kernel memory or crash system.
`while(i < x)`	Upper or lower loop bound: loop termination under users control.
`bcopy(p,q,x)`, `memcpy(p,q,x)`	Length argument for a memory copy operation: corrupt memory.
`copyout(k,u,x)`, `copyin(u,k,x)`	Length argument for user-kernel copy operation: leak information to user or copy unexpected amount of user data into kernel (§ 5.1).
`kmalloc(x)`, `malloc(x)`	Size argument for memory allocator: memory corruption or dangerous style (§ 5.2).

Table 2: Summary of trusting sinks and the attacks allowed if a tainted variable `x` reaches them. The first two sinks are system-independent and hardwired into the checker; the rest are specified on a per-system basis.

find many errors, with few false positives.

Trusting sinks. As summarized in Table 2 the checker flags when unsanitized values reach four types of trusting sinks. The most obvious sinks are arrays whose bounds can be exceeded. Since arrays in C do not have built-in bounds checks, the kernel must prevent the user from reading or writing off either end of the array. The second sink is that if a user value specifies the number of iterations for a loop, that loop can run for "too long," allowing a denial-of-service attack. Finally, the length argument to various copying and allocation routines must be protected. The most straightforward error in this case is that an unchecked length allows an attacker to read or write more kernel data than was expected. Examples of such errors are given in Figure 2 and Figure 7. Allocation based on unsanitized integers is at least bad style, and can also be used to launch subtle memory corruption attacks (see § 5.2). The first two sinks are system-independent; the latter must be specified for each new system.

Thus, the checker looks for approximately $3 * 3 * 3 = 27$ types of security holes: three possible sources of information combined with three checking errors combined with three types of trusting sinks.

Implementation Figure 3 gives a state machine view of the range checker. It has four states:

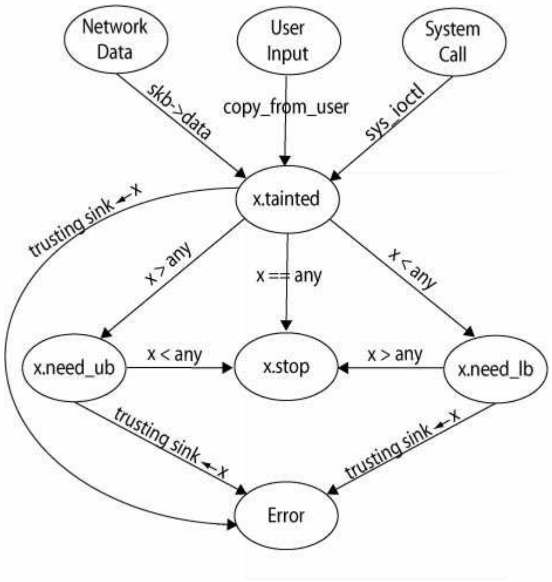

Figure 3: Approximate range checker.

`tainted`, `need_lb` ("need lower bound"), `need_ub` ("need upper bound"), and a special `stop` state. The transition rules are as follows:

- The checker ignores variables in the `stop` state.
- The checker emits an error if a variable in the `tainted`, `need_ub`, or `need_lb` reaches a trusting sink.
- Signed integers read from network data, copied from the user, or from system calls are placed in the `tainted` state; unsigned integers are placed in the `need_ub` state.
- An upper-bound check on a `tainted` variable places it in the `need_lb` state.
- A lower-bound check on a `tainted` variable places it in the `need_ub` state.
- An upper-bound check on a `need_ub` variable places it in the `stop` state.
- A lower-bound check on a `need_lb` variable places it in the `stop` state.
- An equality test on a variable in any state places it in the `stop` state.

Figure 4 gives a textual representation of the range checker state machine. It is written in *metal*, a high-level, state-machine language [8]. During compilation the extension is dynamically linked into our extensible compiler, `xgcc` (based on the GNU `gcc` compiler). After xgcc translates each input function into its internal representation, the checker is applied down every possible execution path in that function.

Typically, extensions use patterns to search for interesting source code features, which, when matched, cause transitions between states. The transition can optionally go to a new state or call into C code. Patterns are written in an extended version of the base language (C), and can match almost arbitrary language constructs such as declarations, expressions, and statements. States can be global (such as the `start` state) or bound to variables or expressions.

In the range checker, each match of a call to `copy_from_user` in the `start` state will place the call's first argument in the `tainted` state and track it separately. After starting a variable, the checker uses patterns to check for ways that the variable can transition out of the `tainted` state. It sets the state of a variable after a bounds check differently depending on whether the check succeeded (using the "`true=...`" notation) or failed (using the "`false=...`" notation). For example, a variable in the `tainted` state that has a lower-bound check done (specified via the pattern "`x > y`") will be put in the `need_ub` (need upper bound) state on the true path of the check and in the `need_lb` (need lower bound) on the false path. The checker stops following a variable once it has seen both an upper- and lower-bound check for that variable.

Retargetting non-network sources and sinks requires giving a routine that indicates whether a given name is a system call, and two text files, listing all untrusted sources and trusting sinks. Figure 5 gives the complete specification for the BSD range checker. It uses the default specification for system calls (that they begin with "`sys_`"). Thus it needs only a list of sources (2) and sinks (13). Linux is only slightly more work: nine sources and fifteen sinks.

The following sections discuss some of the crucial aspects of the checker in more detail: inferring missing actions (§ 3), analysis issues (§ 4), checking subtle rules (§ 5) and results (§ 6).

```
sm range_check {
  // Wild-card variables used in patterns.
  decl any_expr y, z, len; // match any expr
  decl any_pointer v;      // match any pointer
  state decl any_expr x;   // bind state to x

 // Start state. Matches any copy_from_user
 // call and puts parameter x in tainted state.
 start: { copy_from_user(x, y, len) }
        ==> x.tainted
 ;
 // Catch operations illegal on unsafe values.
 x.tainted, x.need_ub, x.need_lb:
    { v[x] } ==>{ err("Dangerous index!"); }
 | { copy_from_user(y, z, x) }
 | { copy_to_user(y, z, x) }
     ==> { err("Dangerous length arg!"); }
 ;
 // Named patterns that match upper-bound
 // (ub) and lower-bound checks (lb).
 pat ub = { x < y } | { x <= y };
 pat lb = { x > y } | { x >= y };

 // Remaining SM code: match code actions that
 // affect tainted variables.  The notation
 //      true=x.<state1>, false=x.<state2>
 // specifies what state to put x in on the
 // true and false branches respectively.
 x.tainted:
   // lower bound check: on the true path the
   // variable needs an upper bound (need_ub);
   // on the false path it needs a lower
   // (need_lb). The other rules are similar.
   lb ==> true=x.need_ub, false=x.need_lb
 | ub ==> true=x.need_lb, false=x.need_ub
 | { x == y } ==> true=x.stop, false=x.tainted
 | { x != y } ==> true=x.tainted, false=x.stop
 ;
 x.need_ub:
    lb ==> true=x.need_ub, false=x.stop
 | ub ==> true=x.stop, false=x.need_ub
 | { x == y } ==> true=x.stop, false=x.need_ub
 | { x != y } ==> true=x.need_ub, false=x.stop
 ;
 x.need_lb:
  lb ==> true=x.stop, false=x.need_lb
 | ub ==> true=x.need_lb, false=x.stop
 | { x == y } ==> true=x.stop, false=x.need_lb
 | { x != y } ==> true=x.need_lb, false=x.stop
 ;
}
```

Figure 4: Stripped down version of the range checker: warns when unchecked data copied from the user is used as an array index or length argument. Some missing checker features: the rules for unsigned variables, the full set of sinks and sources, inherited state (§ 4.2), and the boilerplate needed for interprocedural analysis (§ 4.3)

3 Belief Inference

A weakness of traditional checking approaches is that they rely on some form of specification or hardwired knowledge to encode checking properties. For example, to range check inputs, a strongly typed language might require that the programmer mark all data at all kernel entry points with a "tainted" type qualifier and all sinks with a "not-tainted" qualifier. In addition to being potentially strenuous and invasive, one

Source Function File

```
copyin:1
copyinstr:1
```

Sink Function File

```
copyin:2
copyout:2
copyinstr:2
copyoutstr:2
copystr:2
bcopy:2
bcopyb:2
kcopy:2
bcopyw:2
memcpy:2
copystr:2
fillw:2
malloc:0
```

Figure 5: Complete checker specification for BSD. Two files list (1) the two BSD source functions and (2) the thirteen BSD sink functions. The file format is: `function name ":" argument number`. A trivial edit adds a new source or sink.

real danger with such an approach is that it does not have a safety net to catch omissions. If a parameter is not annotated, it cannot be checked. If a sink is not annotated it will be missed. A key feature of our approach is using code behavior to infer checking properties, thereby allowing us to automatically cross-check them for correctness and completeness or even to eliminate them all together. This section discusses how we use inference to detect: (1) missing sources, (2) missing sinks, and (3) whether network packets are incoming (untrusted) or outgoing (mostly trusted).

3.1 Deriving untrustworthy sources

Obviously, an unchecked, untrusted value can cause exciting trouble. A problem with OS code is that many untrusted values do not come from untrusted sources in a straightforward (analyzable) way. Additionally there are many sources of such untrusted values, making it easy to forget one.

We would like to catch such untrusted input. We can do so by exploiting the fact that code often uses untrusted input in stylized ways. Thus, a value manipulated in such ways implies a belief that the value is dangerous. We can then check this value as we would values produced by known sources. (We could, but do not, also flag the producer of the value as a potentially missing source.)

```
  /* 2.4.9/drivers/telephony/ixj.c:ixj_ioctl */
case IXJCTL_INIT_TONE:
  copy_from_user(&ti, arg, sizeof(ti));
  retval = ixj_init_tone(j, &ti);
  break;
case IXJCTL_INTERCOM_START:
  if (ixj[arg] == NULL)
     return -ENODEV;
  ...
  j->intercom = arg;
  ixj[arg]->intercom = board;
```

Figure 6: Two errors caught by tracking inferred user data; the errors allow an attacker to read or modify arbitrary kernel memory. The checker infers the variable `arg` contains user-supplied data because it specifies the user address for `copy_from_user` and thus flags its subsequent uses as an unchecked array index as errors.

The most effective example of inferring user input has been exploiting a strange but common idiom in OS code: variables that can store either user integers or user pointers depending on context. These variables tend to heavily cluster right where we need them: at interface boundaries where users give input to the OS. This situation is fortunate, since the operating system has very clear, unambiguous ways of treating user pointers. The most common such usage is passing the variable as the user pointer to a routine that copies data between the kernel and user (e.g., `copyin` or `copyout`). The user almost always is the source of this address — the kernel does not spontaneously decide to read or write from arbitrary user addresses. Thus, we know that the variable holding the address will generally contain data from an untrustworthy source and that all uses of the variable as an integer should be checked with the range checker. The key feature of this inference is that we can now check variables that receive untrustworthy data in ways we do not understand. As shown in Section 6, such inference allowed us to catch 12 additional bugs.

Figure 6 gives an example of an error caught using inference. Here, the variable `arg` supplies the source user address to the call `copy_from_user`. We thus know that `arg` generally holds untrustworthy data and should be treated as a tainted variable. This inference allows the checker to catch the two errors on the subsequent branch of the case statement where `arg` is used as an unchecked array index into `ixj` for both a read and a write, neither of which is good.

The inference pass runs before the range checker.

It traverses the compiled code, looking for uses that imply the kernel believes a variable holds user data. It marks the definition of such variable as tainted (i.e., where it was declared, assigned, or passed in). When the range checker subsequently runs, the system places marked variables in the `tainted` state, thereby causing the range checker to follow them.

3.2 Deriving trusting sinks

We also want to find missing sinks. We do so by again using the logic of belief analysis. Given a known source and sink, the normal checking sequence is: (1) OS reads data from unsafe source, (2) checks it, and (3) passes it to a trusting sink. We modified the checker to flag cases where the OS does steps (1) and (2), but not (3). If code reads a value from a known source and sanitizes it, this implies it believes the value will reach a dangerous operation (assuming the programmer is not doing gratuitous sanitization). If the value does not reach a known sink, we have likely missed one. Similarly, we could (but do not) infer missed sources by doing the converse of this analysis: flagging when the OS sanitizes data we do not think is tainted and then passes it to a trusting sink.

We ran the analysis on Linux 2.4.6 and inspected the results by hand to find what we were missing. There were roughly 10 common uses of sanitized input, all but one of which were harmless and did not cause security problems. For example, in a few places, the kernel was using the user value in a switch statement. However, one result we did not expect was that sinks can be missed both by omission and from analysis mistakes — running the check found a place where our inter-procedural analysis had been overly simplistic, causing us to miss a real error (given in Figure 14.)

3.3 Network Data

Generally, remotely exploitable errors are the most dangerous security holes. Our checker flags when data is read from packet headers and then used without checks. One of the key problems we faced was determining if packets were incoming or outgoing. Networking code frequently reads values from outgoing packets (in part because of modularity) as well as incoming packets. We only want to check data read from incoming packets — the data read from outgoing packets is (generally) safe since it is limited to reading packet headers produced by the host OS. Unfortunately, there is no general specification of direction. Instead we must infer one from the code. Otherwise, we could not practically check networking code since our results would be washed out with false positives.

```
/* 2.4.9/drivers/isdn/act2000/capi.c:
        actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
    ...
    msg = skb->data;
    ...
    memcpy(cmd.parm.setup.phone,
          msg->msg.connect_ind.addr.num,
          msg->msg.connect_ind.addr.len - 1);
```

Figure 7: Remotely exploitable error: `msg` points at unchecked network data which can be used to copy arbitrary data onto the stack by overflowing `cmd`.

The Linux kernel keeps track of network data in a structure called "`sk_buff`," with variables of this type commonly named "`skb`." Each field of the structure contains various information about the origins of the data; the pointer to the actual data is in the "data" field. Figure 7 gives an example of a network error. In this code, after dequeueing a network packet `skb` from the receive queue "`card`→`rcvq`," the code sets the pointer, `msg`, to the packet data. It then does an exploitable memory copy where both the length of the copy and the actual copied data come from the the message data. The variable that is overwritten, `cmd`, is on the stack, allowing a malicious user to overwrite a return address and take over the machine.

Distinguishing which `sk_buffs` are being used for incoming and outgoing data is tricky. While in the above example this distinction is clear (pulling `skb` from a list named `rcvq` is a good indication it is incoming data), in general the variety of ad hoc naming conventions would require close to full-blown natural language processing. Further, since we are checking for errors, we do not want to necessarily trust the naming conventions used by programmers.

To solve this problem we use program behavior to infer whether packets are incoming or outgoing. We tried several approaches, but in the end a simple heuristic worked the best: if the checker sees the allocation for the structure, it knows that it is outgoing. Otherwise, it assumes that it is incoming. Using this heuristic, the number of false positives dropped from about 20 to 2.

Our initial attempt to infer direction was both more complex and a failure. Logically, it seems reasonable that the usage of the `skb` structure would help to determine if it was incoming or outgoing. If its fields were read more often than written, the

Expression	Propagation
`q = p`, `memcpy(&q,&p,sz)`	`q` is tainted.
`p+i`, `p-i`, `p*i`	The expression is tainted.
`p++`, `++p`, `p--`, `--p`	`p` remains tainted.
`p→field`	`p`'s fields are tainted.

Table 3: Summary of how a tainted value `p` transitively taints other values.

```
/* OpenBSD 2.9: kern/vfs_subr.c:
     vfs_hang_addrlist */
error = copyin(argp->ex_addr, saddr, ...);
...
i = saddr->sa_family;
if ((rnh = nep->ne_rtable[i]) == 0)
     ...
rn = rnh->rnh_addaddr(...);
```

Figure 8: Inherited state catches this error which allows an attacker to take control of the system by causing the OS to jump to an arbitrary memory location. The attack exploits the hole where user data is (1) read into `saddr`, (2) assigned to `i`, and (3) `i` is used to index into the `ne_rtable` to obtain a function pointer that the kernel jumps to.

checker could infer that the structure was incoming. If the fields were written more often than read, the data would be outgoing. For example, pushing data into an outgoing network packet would look something like:

```
skb->next = skb2;
skb->sock = owner_socket;
skb->data = my_data;
```

whereas pulling data from an incoming network packet would look something like:

```
temp = skb->next;
owner_socket = skb->sock;
msg = (atcapi_msg *)skb->data;
```

Thus, if there were more pulls than pushes performed on an `skb`, the data of that `skb` would be important to the checker. This was not necessarily the case. Drivers often used the fields of outgoing `skb`s for further calculations.

4 Analysis Issues

This section discusses some important practical checker issues: transitive tainting, inherited states, inter-procedural analysis, false positive suppression, false negatives, and ranking errors.

4.1 Transitive tainting

The range checker allows tainted variables to transitively taint other variables. Table 3 summarizes these rules. The simplest: a tainted variable `p` assigned to another variable `q` should cause `q` to be tainted as well. Equivalently, copying `p` onto `q` using a memory copy routine (e.g., `memcpy`) also taints `q`. Adding, subtracting or multiplying a number with a tainted value produces another tainted value. Similarly, incrementing or decrementing a tainted value leaves it tainted. Note, however, that we do not consider the result of dividing a tainted value or computing its modulus to be tainted. Such operations are used by the kernel to perform upper-bound checks: modulus by truncating the tainted value, division by scaling it down. (Note that the division may not scale a tainted value far enough; we do not check for this error.) Finally, if a structure is tainted, its fields are recursively tainted as well (discussed more below). The checker is structured to make adding additional transitivity rules easy.

4.2 Inherited states

While some security violations begin with raw integers coming from user space, it is also an error (and a more common one) that the kernel copies an entire structure from the user and then uses a contained, unchecked value from that structure. Figure 8 gives an example of where missing checks on a structure field allows an attacker to cause the OS to jump to an arbitrary location in memory.

Abstractly, if a structure is tainted, all of its contained data should be as well. To support such "inherited" attributes, we modified the MC system to allow extensions to attach a function to objects that they track. This function controls how any member of that object inherits state from the base object. The function is called on demand when a member is accessed for the first time; it returns the state in which to place the member (if any). The range checker's inherited state function is called whenever a field in a tainted structure is accessed and simply places signed integer members in the `tainted` state and unsigned integers in the `need_ub` state:

```
// Called when field "f" is first
//  referenced in a tainted structure.
int determine_inherited_state(mc_tree f){
    if(mc_is_unsigned(f))
      return need_ub;
```

```
/* 2.4.12-ac3/drivers/char/rio/rioctrl.c:
     riocontrol */
if(copyin(arg, &host, sizeof(host))
                              == COPYFAIL)
   return -1;
if(copyout(p->RIOHosts[host].ParmMapP, arg,
            sizeof(PARM_MAP) ) == COPYFAIL)
   return -1;
  ...

int copyin (int arg, caddr_t dp, int siz) {
  int rv;
  rv = copy_from_user (dp, (void *)arg, siz);
  if (rv < 0) return COPYFAIL;
  else return rv;
}
```

Figure 9: Confidentiality breach caught by computing global untrusted sources. The checker calculates that `copyin` transitively taints its second argument, `host`, and flags when `host` is used as an array index, thereby providing the user with arbitrary information.

```
    else
      return tainted;
}
```

Customized state inheritance makes the checker much cleaner.

4.3 Inter-procedural analysis

The range checker finds errors using both local and inter-procedural analysis (which can span function pointers). While local analysis finds many errors, it misses too many others. In Linux, local analysis found 109 errors, while inter-procedural found 16.

From a client point of view, a key feature of inter-procedural analysis is that it allows the client to only supply the "base" unsafe sources and trusting sinks. The checker then uses inter-procedural analysis to automatically compute all other procedures that could transitively produce or consume data by reaching these base routines. Figure 9 gives a Linux example error caught using inter-procedural source calculations. Here, the `copyin` function, used to emulate BSD's `copyin`, is simply a wrapper for the call to `copy_from_user` . The checker calculates that `copyin` is a source, places `host` in the tainted state, and then flags when `host` is used to index into an array.

Similarly, the checker finds all routines whose argument could reach a trusting sink. If any variable becomes tainted, either because it is a system call parameter, or because it is read from a local or global

```
/* 2.4.9-ac9/fs/ioctl.c:sys_ioctl */
asmlinkage long
sys_ioctl(unsigned fd, unsigned cmd,
                            unsigned arg) {
   ...
   filp = fget(fd);
   if (!filp)
      goto out;
    ...
    error = filp->f_op->ioctl(..., cmd, arg);
...
static int
ip2_ipl_ioctl(..., unsigned cmd,
                          unsigned arg ) {
   ...
   pCh = DevTable[cmd];
   if(pCh)
      COPY_TO_USER(rc, arg, pCh,
                    sizeof(i2ChanStr));
```

Figure 10: Inter-procedural function pointer error that can crash the system or breach confidentiality. The value `cmd` comes from the user and is passed unchecked to the function pointer call `filp→f_op→ioctl`. The checker emits an error since the pointer can potentially point to `op2_opl_ioctl` which uses `cmd` unsafely.

sink, the checker will follow all calls this data is passed to, flagging if it is used incorrectly. The security hole in Figure 1 is one example caught by this analysis.

Inter-procedural analysis works as a two-pass process. The first pass computes summaries of all calculated sources and sinks as follows:

1. It runs an MC extension that emits a callgraph for the entire OS.

2. It then supplies the list of root sources and sinks to an MC relaxation program, which uses the emitted global callgraph to compute the transitive set of functions (and their arguments) that could reach these routines.

3. These calculated sources and sinks are emitted in two text files as three tuples giving: (1) the function name, (2) the argument, (3) the path to the source or sink (for error reporting).

The second pass uses these summaries to flag errors. At each function call site it computes whether the call produces new tainted variables or consumes existing ones (or both). In the first case it follows the variables. In the second it emits an error message.

The checker also follows function pointers in addition to simple calls. During the first pass above, it records all function pointer assignments or initializations. During the second pass, it checks function pointer calls by seeing if any routine the pointer could reference is on a summary list and emits an error or taints the argument as needed.

Figure 10 gives an error caught this way. Here a tainted value `cmd` is passed to a function pointer `filp→f_op→ioctl`. The checker compares the functions the pointer can reference against the list of functions whose arguments reach a trusting sink. It emits an error since the pointer can reference a routine (`op2_opl_ioctl`) that uses the `cmd` argument unsafely.

4.4 False positives

We generally write checkers by first building a simple, weak version. If this finds enough bugs to be interesting we make it perform deeper analysis. Part of this process is modifying the checker to reduce false positives. Usually, false positives will come from a small number of sources, which can be suppressed with targeted checker modifications. In the case that there are only a few errors caused by a single feature we usually do not change the checker. Instead we use "history" to ensure that we only inspect an error or false positive once. For error reporting, we store errors in annotated logs. These logs represent errors as a tuple of features that are relatively invariant under edits (the file and function containing the error, the variables and function calls involved, etc.). We use these tuples to automatically classify errors and false positives that last to later releases. Below we discuss the major causes of false positives.

The first is code that does simultaneous upper- and lower-bounds checks on signed integers by casting them to an unsigned value and doing an upper-bound check. For example:

```
/* equivalent checks. */
if(userlen < 0 || userlen > MAX)
   return -1;
if((unsigned)userlen > MAX)
   return -1;
```

The checker did not originally account for this idiom, but once it did, about 10 false positives were eliminated.

The second source was because the inference of what data came from the user was often too general. The checker would see that a field of a structure came from the user and treat the whole structure as tainted instead of just that field. While this counted for six of the false positives, it was beneficial not to change the checker because when it was correct, the errors were quite severe.

The third source came from the fact that if a subroutine bounds checked a tainted variable, we would not propagate the bounds check up to the caller. (Though we do propagate the value down to all routines the checking routine calls.) There were 5 false positives of this type.

The final source of false positives were range-check errors on code paths that could only be executed by the super-user. We counted these as minor errors rather than false positives since they were, at the very least, bad style.

4.5 False negatives

While the checker finds a large number of different types of bugs; it misses others. We discuss the main (known) categories below.

1. As stated in Section 2, the checker considers a value sanitized after being checked against any value rather than the right value. We are currently exploring the use of belief analysis to derive which bounds users believe a pointer or routine requires as a precondition.

2. The system only tracks values within a given code path. It will miss errors where one system call stores a tainted value in a data structure and a different system call subsequently extracts the value and uses it.

3. The system loses the states of variables stored inside structures when they go across function calls. For example, if a tainted value is stored in `cmd→len` and `cmd` is passed to function `foo(cmd)`, the system will not remember that `cmd→len` is tainted while analyzing `foo`.

4. If a parameter is inferred to come from the user the system does not propagate that information to the caller. For example, if while analyzing `bar(int arg)`, the checker determines that `arg` must come from the user, callers of `bar` do not benefit from this information.

The last three limitations are mainly due to the MC system itself. It is currently being reimplemented around a core set of more powerful analyses. Extensions such as the range checker should (mostly) transparently benefit from these improvements.

4.6 Ranking errors

Given a large number of errors, ranking errors is as important as eliminating false positives. We use both

```
/* 2.4.6/drivers/net/wan/sdla.c:sdla_xfer */
if(copy_from_user(&mem, info, sizeof(mem)))
     return -EFAULT;
if (read) {
     temp = kmalloc(mem.len, GFP_KERNEL);
     if (!temp)
         return(-ENOMEM);
```

Figure 11: Oblique checking: the driver uses `kmalloc` as an implicit bounds check.

generic and checker-specific ranking. The generic ranking stratifies errors based on how easy they are to diagnose as well as how likely they are to be false positives. Our ranking places local errors over global ones, errors that span few source lines and conditionals over those that span many, errors that do not involve aliasing over those that do.

Our checker-specific ranking marks minor error patterns to distinguish them from potentially more serious ones. The main such demotion was errors involving allocation functions. Kernel allocation functions have a fixed upper-bound on the amount of memory they allocate at one time. There were many cases where the code was (possibly unintentionally) using the allocator to act as an implicit bounds check by relying on it to fail and return a `NULL` pointer when a large value was passed in. (Figure 11 gives an example.) While errors of this type are not exploitable security holes, relying on the allocator seems to be at least bad form since the size checked is an artifact of the allocator implementation. Further, rather than returning an error code saying that the user-specified value was too large, the caller of the allocator would most likely return an error code saying that the kernel was out of memory. This would not be the case and could cause problems further along.

5 Enforcing Obscure Rules

Many security rules are poorly understood and erratically obeyed. This section shows two of our best examples of how checkers can give significant practical leverage by allowing one person to understand an obscure rule and write a checker that is imposed on all code. The first comes from an attack using user-kernel data movement routines, the second from overflow of fixed-size arithmetic.

5.1 The length-field copy attack

The user-kernel data movement routines in different operating systems share a common underlying assumption that allows attackers to breach confidentiality or integrity by reading or writing much more memory than was anticipated [4]. The attack was unknown to many seasoned kernel implementers — to the extent that they debated its validity until shown a specific attack. The attack works on both BSD and Linux; we assume it works on other OSes as well. A typical type signature of these routines would be:

```
/* BSD */
int copyout(void *kern, void *usr,
                        unsigned len);
/* Linux */
int copy_to_user(void *usr, void *kern,
                        unsigned len);
```

```
/* 2.4.1/kernel/sysctl.c:do_sysctl_strategy */
int len;
...
  get_user(len, oldlenp);
  if (len) {
     if (len > table->maxlen)
         len = table->maxlen;
     if(copy_to_user(oldval, table->data, len))
         return -EFAULT;
```

Figure 12: A rare kernel security hole that allows an attacker to breach confidentiality. Upper-bound check on the integer `len` but no lower-bound allows an attacker to copy nearly arbitrary amounts of kernel data back into user space.

These routines explicitly work with untrusted pointers and ensure that the specified user-virtual address range $[user, user+len)$ is valid: both completely contained within the user's address space, and, more strongly, all the memory contained has some valid mapping. Naively, it would then seem that if the kernel does an upper-bound check on `len` to ensure that it is smaller than a maximum size that there could be no way that this routine could be circumvented.

Unfortunately, as Figure 12 shows this is not sufficient. This case is interesting in that it was an error in the core kernel, rather than a driver and serves to illustrate how widely misunderstood security rules can be. The attack involves two user-supplied variables: the signed integer variable `len` and the pointer `oldval`. The code first reads in `len` using the macro `get_user`. It then does an upper-bound check on `len` (but no lower-bound check) and then passes it as the unsigned length argument to `copy_to_user`. Unfortunately, if `len` is negative, it will pass the bound check but become a large positive value when passed to `copy_to_user`. The most straightforward attack is

```
/* 2.4.9/drivers/net/wan/farsync.c:fst_ioctl */
if(wrthdr.size + wrthdr.offset > FST_MEMSIZE)
    return -ENXIO;
if(copy_from_user (card->mem + wrthdr.offset,
                  ifr->ifr_data, wrthdr.size))
    return -EFAULT;
```

Figure 13: Overflow error that allows an attacker to potentially take control of the system. Large values of `wrthdr.size` and `wrthdr.offset` can cause the expression to "wrap around" to a small number, vacuously passing the range check but allowing an attacker to overwrite any region of kernel memory with arbitrary values.

to pass the kernel a well-chosen pointer address for `oldval` and a negative value for `len` such that the range $[oldval, oldval + unsigned(len))$ is a valid user address range. In this case, the attacker will be able to copy out near arbitrary amounts of kernel memory. (The straightforward fix is to make `len` unsigned.)

5.2 Overflow fun

While the need to bound values is conceptually simple, programmers often seem to forget the behavior of fixed-size arithmetic. C has unsurprising rules for integer overflow: positive signed values that overflow become negative, and unsigned values that overflow wrap around toward zero. Figure 13 gives a typical error. Here, the programmer attempts to do an upper-bound check on `wrthdr.size` and `wrthdr.offset` using the expedient method of adding them together and checking that their sum is less than `FST_MEMSIZE`. Unfortunately, because the expression can overflow and then "wrap around" to a small value, an attacker can cause this check to succeed even when the sum of the variables is very large. The hole lets the attacker overwrite any region of kernel memory with a near-arbitrary amount of data they supply (via `ifr→ifr_data`).

Figure 14 gives another, more subtle overflow error. Here the variable `input` is read from the user. The kernel then allocates a temporary buffer of size `input.path_len + 1` bytes and then copies `input.path_len` bytes of user data into it. As discussed in Section 4.6 an allocation call with a large value usually acts as an implicit bounds check, since `kmalloc` will return a `NULL` pointer for large allocation requests. Unfortunately, overflow allows an attacker to defeat the implicit check in the example. If `input.path_len` holds the maximum unsigned value, the addition of 1 will cause the value to wrap to 0.

```
/* 2.4.9-ac7/fs/intermezzo/psdev.c:
        presto_psdev_ioctl */
err = copy_from_user(&input,arg,sizeof(input));
...
input.path = kmalloc(input.path_len + 1,
                       GFP_KERNEL);
if(!input.path)
      return -ENOMEM;
error = copy_from_user(input.path, user_path,
                              input.path_len);
```

Figure 14: Overflow-induced under allocation that allows an attacker to breach kernel integrity. If `path_len` equals the largest possible unsigned value the addition of 1 will wrap it to zero. A call to `kmalloc(0)` returns a non-nil pointer to a (small) amount of kernel memory, which the subsequent `copy_from_user` can be used to overflow, corrupting large amounts of kernel memory.

However, a zero-byte allocation call to `kmalloc` will return a non-nil pointer to a (small) buffer, which the subsequent copyin will then exuberantly overwrite with (say) 4GB of user data.

The checker catches these errors using a very primitive approach: it treats all upper- and lower-bound checks that involve arithmetic on user data as vacuous. While this approach causes false alarms when a clever programmer does fancy bounds checks correctly, empirically such cases are negligibly rare.

6 Range Checker Results

Table 4 tabulates the range checker errors for both Linux (125 errors) and OpenBSD (12 errors). Errors are classified into five categories of decreasing severity:

1. Gain control of the system. A malicious user could use these holes to gain control of the system by jumping to arbitrary code or changing permissions to become the "superuser." At the very least, these holes allow an attack to crash the kernel.

2. Breach integrity by corrupting kernel memory. These errors were most often a result of overflow math errors that caused the kernel to under-allocate memory, which then caused subsequent writes to corrupt adjacent memory locations.

3. Read arbitrary memory. These include both confidentiality breaches where attackers could read information they were not supposed to

Violation	Linux		OpenBSD	
	Bug	Fixed	Bug	Fixed
Gain control of system	18	15	3	3
Corrupt memory	43	17	2	2
Read arbitrary memory	19	14	7	7
Denial of service	17	5	0	0
Minor	28	1	0	0
Total	125	52	12	12

Table 4: Error breakdown for Linux and OpenBSD.

see as well as system crashes due to attackers causing the system to read invalid memory addresses.

4. Denial of Service. While not serious enough to take over the system, these errors could be exploited to make the kernel loop for a attacker-controlled amount of iterations.

5. Minor. These were errors where the kernel code was using an allocation function as an implicit range check in user input. Errors also fell into this category if the error could only be exploited by the superuser — it is unlikely that the superuser would want to perform a buffer overflow attack, but he could trigger it by accident and crash the machine.

Our main caveat is that the result breakdown should be regarded only as approximate. The sheer number of errors and the fact that we are not core kernel implementers makes it possible that we mis-categorized some.

Table 5 summarizes our experimental results. There were two results that surprised us. First, severe errors seem just as common as minor ones. On Linux in particular, the number of errors in the worst error categories roughly equal those in the last three. We had expected that the severe errors would be by far the most rare — these rules should be the most widely known, and the programmers the most attentive. Second, most bugs were local. Our initial checker was local-only; we had expected that making it inter-procedural would dramatically increase the number of bugs found.

Inference of user input worked well on Linux (though it did not find OpenBSD errors). To get a feel for the number of checks performed (and hence the error rate) we counted the approximate number of variables checked in Linux 2.4.12 — there were roughly 3500 such variables (we count a structure with multiple fields as one variable). If we make the crude assumption that all errors found by our checkers in earlier versions of Linux would have otherwise persisted until 2.4.12 this gives an error rate of roughly 1 out of 28 variables being mishandled (125 errors / 3500 checks), which points to a strong need for automatic checking and programmer education. OpenBSD had a better but still high error rate: roughly 1 out of 50 variables mishandled (594 checks / 12 errors). However, note that all of these errors were serious ones that could allow an attacker to crash or take control of the system.

Result gathering and validation. The first runs of the range checker were over Linux 2.4.5. As we improved the checker and found more bugs we reported them to the Linux Kernel mailing list. The reports came roughly in batches every few weeks over a period of about four months, lasting to version 2.4.12-ac3, which was released two weeks before this paper was submitted. However, while errors come from multiple releases each bug is unique — we count each error exactly once rather than once for each release in which it appeared. Many of the reports resulted in kernel patches just a few hours after their submission. We used these patches to verify that the security holes were real, but the lack of a patch did not necessarily mean that the report was false. Some bugs were minor and fixing them introduced the possibility of adding new bugs. If a report was false, the kernel developers would tell us and explain why. We focused mainly on Alan Cox's intermediary versions of Linux rather than Linus Torvalds' because Cox followed a more frequent release schedule. Most of our checking focus was on the versions 2.4.5 — 2.4.12, though we did check one outlying version (2.4.1).

While the bulk of our results are on Linux, the range checker is easily adapted to other systems. We ran the checkers on OpenBSD 2.9 and found the range checker to be quite successful. Here, errors were validated by submitting them to Costa Sapuntzakis, a local BSD hacker, who in turn submitted security advisories for us. All of those bugs resulted in kernel patches.

Category	Linux (2.4.1, 2.4.5 — 2.4.12-ac3)	OpenBSD 2.9
Total bugs	125	12
Local bugs	109	12
Global bugs	16	0
Bugs from inferred user integers	12	0
Network errors	1	0
False positives	24	4
Number of sources	9	2
Number of sinks	15	13
Number of inferred sinks	4	0
Number of variables checked	roughly 3500 (on 2.4.12)	594

Table 5: Summary of experimental results.

Thus, the errors went through a rigorous examination process. While there are likely to be some false positives, we would be surprised if this number was more than 5%.

7 Other checkers

This section briefly discusses two other security checkers.

7.1 A user-pointer checker

The user-pointer checker warns when pointers copied from the user are dereferenced. If a pointer originates from an untrustworthy source, the kernel must use `copy_from_user` to access the data referenced by that pointer. The kernel cannot simply dereference the pointer.

We have previously presented a user-pointer checker that followed system call parameters [8] and behaviorally inferred which pointers kernel code believed were user pointers and which it believed were kernel pointers [9] (similar to the inference in § 3.1). A weakness in this past checker is that it did not follow data explicitly copied from user space. We extended it using the machinery developed for the range checker.

The user-pointer checker uses the range checker's untrusted sources to mark all values copied from an untrusted source as tainted. It also uses state inheriting to recursively mark pointers contained in tainted structures as tainted. It then uses inter-procedural analysis to follow tainted pointers, flagging any dereference. While its analysis is as deep, the extension is much simpler than the range checker since in most operating systems there are no checks that can make tainted pointers safe.

When we ran the user-pointer checker on Linux 2.4.9-ac10 it found 6 errors with 7 false positives. These errors would be missed by our prior work. The false positives were mainly due to a single special-case check convention provided by the `get_user()` macro. We did not take the trouble to modify the checker to eliminate them.

The most common error was copying a structure from the user that contained a pointer and then dereferencing it. Figure 15 gives a representative error that allows an attacker to crash the machine. Here a structure (`buf`) is copied from user space. A pointer field it contains (`buf.win_info.handle`) is then passed as an argument to a routine that dereferences it.

```
/* 2.4.9-ac10/drivers/pcmcia/ds.c */
if (cmd & IOC_IN)
    copy_from_user(&buf, arg, size);
...
ret = pcmcia_get_mem_page(buf.win_info.handle,
                          &buf.win_info.map);
...
/* drivers/pcmcia/cs.c:pcmcia_get_mem_page */
int pcmcia_get_mem_page(window_handle_t win,
                        memreq_t *req)
    if ((win == NULL)
    || (win->magic != WINDOW_MAGIC))
```

Figure 15: User pointer error: `buf` was copied from the user and then the field `buf.win_info.handle` (a pointer) is passed to the `pcmcia_get_mem_page` routine which promptly dereferences it.

7.2 Marking user-triggered errors

Our prior work checked mainly for non-security errors. A typical run can find hundreds or even thousands of such errors in a system the size of Linux.

```
/* 2.4.4-ac8/drivers/block/cciss.c:
     cciss_ioctl */
   if(iocommand.buf_size > 128000)
     return -EINVAL;
   if(iocommand.buf_size > 0) {
     buff =  kmalloc(iocommand.buf_size, ...);
     if( buff == NULL)
         return -EFAULT;
   }
   if (iocommand.Request.Type.Direction
                              == XFER_WRITE) {
     if (copy_from_user(...))
         return -EFAULT; /* lost buff! */
```

Figure 16: Lost memory security hole: a malicious user can cause the driver to leak up to 128K of memory (pointed to by buff) on each call.

```
/* 2.4.4-ac8/net/atm/common.c:atm_ioctl */
   spin_lock (&atm_dev_lock);
   ...
   case SIOCOUTQ:
     ...
     ret_val =  put_user(...);
     goto done;
   case SIOCINQ:
     ...
     ret_val = put_user(...);
     goto done;
```

Figure 17: Potential attacker-initiated deadlock: because they occur with a spin lock held, each call to put_user can cause the system to deadlock. This single routine had 19 such errors.

```
/* 2.4.4-ac8/drivers/block/cciss.c:
                      cciss_ioctl  */
  if (copy_to_user(...)) {
      cmd_free(NULL, c);
      if (buff != NULL) kfree(buff);
      return( -EFAULT);
  }
  if (iocommand.Direction == XFER_READ)
      if (copy_to_user(...)) {
          cmd_free(NULL, c);
          kfree(buff);
      }
  cmd_free(NULL, c);
  if (buff != NULL) kfree(buff);
```

Figure 18: Double-free security hole: the first copy_to_user correctly deallocates its storage and returns with an error. The second appears to have omitted a return statement, thus causing control to fall through and double free both c and buff.

Thus, in practice, many errors are not fixed. Unfortunately, if any of these unfixed errors can be triggered by a user, they are a security risk. An attacker could use them as the basis of a difficult-to-track denial of service attack by deliberately triggering them to cause resources leaks or even machine crashes.

The extension in this section flags errors that seem to be exploitable by the user. It runs before normal checking extensions and marks paths reachable from the user as exploitable paths. Later, when a standard checker emits an error, a custom printing routine checks for the "exploitable" annotation and if it there marks the emitted error as a security risk. These errors are ranked above all others.

The extension heuristically marks two paths as exploitable paths: (1) those that contain calls to copy data between kernel and user space and (2) those that contain any call to a function that checks permissions. The underlying rationale for the heuristic is that such calls are generally reachable by the user since the system does not copy data or check permissions on a whim. Further the user can control which path is taken: the successful branch by satisfying necessary checks, the failed branch by passing in a bad pointer or by asking for an operation that they lack permissions for.

The extension found three classes of errors:

1. Ten places in Linux 2.4.4-ac8 where storage was not released in response to user errors. A malicious user could easily cause the kernel to run out of memory by repeated invocations. Figure 16 gives one of the more egregious leaks.

2. Sixty-two unique errors in various incremental versions of Linux 2.4.4 where a call to a blocking, user-data copy routine lets an attacker potentially deadlock the system by giving the OS a pointer to memory that has been paged to disk, thereby causing it to sleep with a lock held. Figure 17 gives two such errors from a routine (atm_ioctl) that had 19 in total.

3. Two double-free errors in 2.4.4-ac8. Figure 18 gives both; note they are in the same routine (cciss_ioctl) from Figure 16.

The extension also reinforces the point of Section 5: checkers are an effective way to enforce poorly-understood rules. In this case, one of the errors annotated by the extension had appeared in a prior paper without us realizing that it was a security risk!

8 Related Work

There are many methods for finding software errors. The most widely used, testing and manual inspection, suffer from the exponential number of code paths in real systems and the erratic nature of human judgment. Below, we compare our approach to other methods of finding security errors in software: high-level compilation, dynamic checking, and type systems.

There has been much recent work in automatic static detection of security holes. Bishop and Dilger [2] were among the earliest, and describe a system that uses global information to detect "time-of-check-to-time-of-use" (TOCTTOU) race conditions in privileged Unix applications. More recently, there has been work on finding information leaks [12], intrusion detection [17], and a lot of attention paid to detecting buffer overflows [6, 11, 13, 16, 18]. More generally many projects have embedded hard-wired application-level information in compilers to find errors [1, 2, 3, 7, 14, 18]. At a low level, our checkers find different error types than this prior work. At a higher level, these projects find a fixed set of errors, whereas we show that a general, extensible framework can be used instead, allowing the checking of a broad range of system-specific properties.

In the context of dynamic analysis, the "tainting" feature in Perl [19] is a widely-used, effective way of finding unchecked uses of untrustworthy data. In a sense the approaches are complementary. Such dynamic monitoring serves as a hard end-to-end check that no error is made. Static analysis allows these errors to be caught at compile time without runtime overhead, rather than potentially crashing the system. Pragmatically, the effort of adding such dynamic information to a systems programming language seems much greater than writing checkers.

Language type systems probably find more bugs on a daily basis than any other automatic approach. However, many program restrictions—especially temporal or context-dependent restrictions—are too rich for an underlying type system or are simply not expressed in it. While there has been some work on richer frameworks such as TypeState [15], Vault [5], and aspect-oriented programming [10], these still miss many systems relations and require programmer participation. Further, from a tool perspective, all language approaches require invasive, strenuous rewrites to get results. In contrast, our approach can precisely check properties without requiring the use of a specific language or ideology for code construction. In our opinion, no one would use a tool that required the number of annotations needed by a type system; calling a tool a "type system" is not enough to make such overheads palatable.

One feature to note about stronger type checking is that it is a mistake to think that the errors found in this paper would have been "solved" if C provided dynamic bounds checks. Such checks would cause a runtime exception when violated. Exceptions are mishandled notoriously often. A common example is exception handling code that does not reverse all necessary effects (e.g., releasing locks, unpinning memory, decrementing reference counts), which at the very least allows denial-of-service attacks.

Finally, we have noticed that an extension-based approach for checking has several advantages over annotation-based approaches (such as type systems). One advantage is that extensions significantly reduce the cost of specification by combining analysis with ad hoc knowledge. A good example of this is the function in Section 2 used to indicate if a routine is a system call. Rather than having to annotate every system call parameter in the source code (possibly missing some), one can simply write a fixed-cost extension that automatically marks all routines beginning with "`sys_`" as having tainted arguments. A second advantage is that extensions can use customized analysis to both infer checking information from code and to catch when they miss check-related actions. Passive annotations can do neither.

9 Conclusion

This paper has shown how to use programmer-written compiler extensions to catch security errors. We have presented one checker and sketched two others. The extensions worked well in practice. We found well over 100 errors in two systems (Linux and BSD), some in core kernel code. Kernel implementers have patched over 50 of the errors in response to our reports.

Our most developed checker, the range checker, used novel techniques both to eliminate the need to specify some checking properties and to detect incomplete (or incorrect) specifications. We used such inference in three places. First, to detect missed sources, the range checker looked for other uses of input that suggest that it comes from the user. Second, to find missed sinks, the checker looks for places where data comes from a known, unsafe source, is checked, but is then not used for anything for which the checker knows to look. Third, to determine what data comes from an incoming (rather than outgoing) network packet the checker uses code analysis rather than specifications. We hope to extend the checker to allow easy addition of arbitrary data types (such as strings) that require preconditions be met before use.

We believe the use of system-specific static analysis is a general approach to finding security errors. A practical advantage is that it allows automatic enforcement of obscure, poorly understood rules.

10 Acknowledgments

We thank Costa Sapuntzakis for getting us started with BSD and for verifying our results. We thank Alan Cox for explaining how capability checks affect the results of our checker and for pointing out the potential errors with arithmetic overflow. Andy Chou implemented much of the MC system's support for inter-procedural analysis; Seth Hallem did numerous system fixes. Godmar Back and Seth Hallem gave valuable proof-reading assistance. Wilson Hsieh's extensive, last minute comments greatly helped the presentation.

References

[1] A. Aiken, M. Faehndrich, and Z. Su. Detecting races in relay ladder logic programs. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 1998.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

[3] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[4] Chris Evans chris@scary.beasts.org. Personal communication. Negative parameter passed to `copy_to_user` or `copy_from_user` allows exposing or overflowing arbitrary kernel memory, April 2001.

[5] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[6] N. Dor, M. Rodeh, and S. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *8th International Symposium on Static Analysis (SAS)*, pages 194–212, July 2001.

[7] P. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 1–14, New York, NY, 1999.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[9] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[11] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium, Washington, D. C.*, August 2001.

[12] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142, October 1997.

[13] Jon Pincus. Personal communication. Developing a buffer overflow checker in PREfast (a version of of PREfix)., October 2001.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[15] R E Strom and S Yemini. TypeState a programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 1:157–171, January 1986.

[16] J. Viega, J.T. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Annual Computer Security Applications Conference*, 2000.

[17] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[18] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, February 2000.

[19] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly, 1991.