

Introduction to Reversing with Z3

RPISec

Avi Weinstock (aweinstock)

December 3, 2019

```
static inline void probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) < 0)
        return 0;

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == ((BN_ULONG)-1)) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) != 0) {
                delta += 2;
            }
        }
    }
}
```

Overview

- ▶ What are SAT/SMT/Z3?
- ▶ Solving MBE lab1A with Z3
- ▶ Solving a Cyberseed RE challenge with Z3

```
static int prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) &
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

What is SAT?

- ▶ SAT is the boolean SATisfiability problem
- ▶ e.g. "Does the formula $(x \vee \neg y \vee z) \wedge (\neg x \vee y)$ have a satisfying assignment?"
- ▶ Brute forceable in $O(2^n)$ by trying all combinations of $\{0,1\}$ for all variables
- ▶ NP-Complete
 - ▶ Not known to be subexponentially solvable in general
 - ▶ Many problems expressible as SAT

```
static prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid infinite loop. */
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        } else {
            size_limit = ((BN_ULONG)1) << bits;
        }
        maxdelta = size_limit;
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) <= 1) {
                delta += 2;
            }
        }
    }
}
```

What is SMT?

- ▶ SMT is Satisfiability Modulo Theories
- ▶ "Does $(f(x, y) \vee z) \wedge (\neg g(x) = f(x, x))$ have a satisfying assignment?" (QF-EUF)
- ▶ "Does $2 * x + y \leq z \wedge x + 3 * y \geq z$ have a satisfying assignment?" (QF-LIA)
- ▶ Allows more compact translation of problems, e.g.
 - ▶ $x = 1 \vee x = 2 \vee x = 3 \vee \dots \vee x = 99 \vee x = 100$ (SAT)
 - ▶ $1 \leq x \wedge x \leq 100$ (SMT)
- ▶ Also NP-Complete

```
static int prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        /* Only one size limit. */
        /* Valid under test behavior. */
        size_limit = ((BN_ULONG)0) - get_word(rnd);
    } else {
        /* Two size limits. */
        size_limit = ((BN_ULONG)0) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
        maxdelta = size_limit;
    }
    delta = 0;
    if (!is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) ||
                (primes[i] == 2 && (rnd_word & 1) == 0)) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    }
    return 1;
}
```

Why are SAT/SMT useful if they're hard to solve quickly?

- ▶ Not all problems are as hard as the hardest ones
 - ▶ 2-SAT (each clause having at most 2 variables) is polytime solvable
 - ▶ Monotone circuits (only ANDs and ORs, no NOTs) are polytime solvable
- ▶ It's often possible to prune the search space
 - ▶ e.g. $x \vee \varphi(a, b, c, \dots)$ is solvable regardless of φ because $x = 1$ cancels out that subterm
- ▶ Algorithms like DPLL and CDCL make use of partial structure to solve some instances faster than others
- ▶ SMT can make use of the rules for the extra types of symbols to prune the search space at a higher level

```
static_prime(BIGNUM *rnd, int bits) {
    int i;
    uint64_t mod;
    BN_ULONG mask2 = primes[NUMPRIMES - 1];
    clear_is_single_word = bits <= BN_BITS2;

again:
    if (BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint32_t)mod;
        /* if it fits into a single word then we
         * additionally don't want to exceed that many bits. */
        if (bits <= BN_BITS2) {
            if (bits == BN_BITS2) {
                /* Avoid undefined behavior. */
                size_limit = (((BN_ULONG)0) - get_word(rnd));
            } else {
                size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
            }
            maxdelta = size_limit;
        }
        delta = 0;
    }

    BN_ULONG rnd_word = get_word(rnd);

    /* In the case that the candidate prime is a single word then
     * we check that:
     * 1) It's greater than primes[i] because we shouldn't reject
     *    3) as being a prime number because it's a multiple of
     *    5) as being a prime number because it's a multiple of
     *    7) as being a prime number because it's a multiple of
     *    11) as being a prime number because it's a multiple of
     *    13) as being a prime number because it's a multiple of
     *    17) as being a prime number because it's a multiple of
     *    19) as being a prime number because it's a multiple of
     *    23) as being a prime number because it's a multiple of
     *    29) as being a prime number because it's a multiple of
     *    31) as being a prime number because it's a multiple of
     *    37) as being a prime number because it's a multiple of
     *    41) as being a prime number because it's a multiple of
     *    43) as being a prime number because it's a multiple of
     *    47) as being a prime number because it's a multiple of
     *    53) as being a prime number because it's a multiple of
     *    59) as being a prime number because it's a multiple of
     *    61) as being a prime number because it's a multiple of
     *    67) as being a prime number because it's a multiple of
     *    71) as being a prime number because it's a multiple of
     *    73) as being a prime number because it's a multiple of
     *    79) as being a prime number because it's a multiple of
     *    83) as being a prime number because it's a multiple of
     *    89) as being a prime number because it's a multiple of
     *    97) as being a prime number because it's a multiple of
     *    that's true. */
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
        if ((mods[i] + delta) % primes[i] == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }

} else {
    for (i = 1; i < NUMPRIMES; i++) {
        /* check that rnd is not a prime and also
         * that gcd(rnd-1, primes) = 1 (except for 2) */
        if (((mods[i] + delta) % primes[i]) != 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```

What is Z3?

- ▶ SAT & SMT solver developed and maintained by Microsoft Research
- ▶ Libre and Open Source (MIT Licensed)
- ▶ C++, with python bindings (`pip install z3-solver`)
- ▶ Based on the CDCL algorithm

```
static int prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) &
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit = BN_ULONG_MAX;
        /* Avoid undefined behavior. */
        size_limit = (((BN_ULONG)0) - get_word(rnd));
    } else {
        size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
        maxdelta = size_limit;
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

Using Z3 on small examples

► $(x \vee \neg y \vee z) \wedge (\neg x \vee y)$

```
import z3
solver = z3.Solver()
x, y, z = z3.Bools('x y z')
solver.add(z3.And(z3.Or(x, z3.Not(y), z), z3.Or(z3.Not(x), y)))
if solver.check().r == 1:
    print(solver.model())
```

► $2 * x + y \leq z \wedge x + 3 * y \geq z \wedge z > 1$

```
import z3
solver = z3.Solver()
x, y, z = z3.Ints('x y z')
solver.add(2*x+y <= z)
solver.add(x+3*y >= z)
solver.add(z > 1)
if solver.check().r == 1:
    print(solver.model())
```

```
static inline prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) < 0)
        return 0;

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;
}
```

```
loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
            }
        }
    }
}
```

MBE Lab1A - Just Running It

```
avi@aweinstock-debian-ii:~/Documents/cloned-repos/MBE/src/lab01$ ./lab1A
```

```
+-----+
|----- RPISEC -----|
|+ SECURE LOGIN SYS v. 3.0 +|
|-----|
|~- Enter your Username: ~-|
|-----|
```

username

```
+-----+
| !! NEW ACCOUNT DETECTED !!|
|-----|
|~- Input your serial: ~-|
|-----|
```

password

```
avi@aweinstock-debian-ii:~/Documents/cloned-repos/MBE/src/lab01$ echo $?
```

1

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
```

```
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mod[i] = delta % primes[i]) <= 1) &&
                delta % 2) {
```


MBE Lab1A - Username Entry

```
0x08048b69 c70424738d04, mov dword [esp], str..-----, ; [0x8048d73:4]=0x2d2d2d2e ; ",-----",
0x08048b70 e89bfcffff call sym.imp.puts ; int puts(const char *s)
0x08048b75 c70424918d04, mov dword [esp], str,RPISEC ; [0x8048d91:4]=0x2d2d2d7c ; "|----- RPISEC -----|"
0x08048b7c e88ffcffff call sym.imp.puts ; int puts(const char *s)
0x08048b81 c70424af8d04, mov dword [esp], str.SECURE_LOGIN_SYS_v._3.0 ; [0x8048daf:4]=0x53202b7c ; "|+ SECURE LOGIN SYS v. 3.0 |+|"
0x08048b88 e883fcffff call sym.imp.puts ; int puts(const char *s)
0x08048b8d c70424cd8d04, mov dword [esp], str. ; [0x8048dcd:4]=0x2d2d2d7c ; "|-----|"
0x08048b94 e877fcffff call sym.imp.puts ; int puts(const char *s)
0x08048b99 c70424eb8d04, mov dword [esp], str.Enter_your_Username ; [0x8048deb:4]=0x202d7e7c ; "|~- Enter your Username: ~-|"
0x08048ba0 e86bfcffff call sym.imp.puts ; int puts(const char *s)
0x08048ba5 c70424098e04, mov dword [esp], str. ; [0x8048e09:4]=0x2d2d2d27 ; "'-----'"
0x08048bac e85ffcffff call sym.imp.puts ; int puts(const char *s)
0x08048bb1 a160b00408 mov eax, dword [obj.stdin] ; [0x804b060:4]=0
0x08048bb6 89442408 mov dword [local_8h], eax
0x08048bba c74424042000, mov dword [local_4h], 0x20 ; [0x20:4]=-1 ; 32
0x08048bc2 8d44241c lea eax, [local_1ch] ; 0x1c ; 28
0x08048bc6 890424 mov dword [esp], eax
0x08048bc9 e802fcffff call sym.imp.fgets ; char *fgets(char *s, int size, FILE *stream)
```

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) &
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }

        /* 2) that it's not a multiple of a known prime. We don't
         * check that rnd-1 is also coprime to all the known
         * primes because there aren't many small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mod[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mod[i] + delta) % primes[i]) == 0) {
                delta += 2;
            }
        }
    }
}
```

MBE Lab1A - Serial Entry

```

0x08048bce c70424738d04. mov dword [esp], str.----- ; [0x08048d73;4]=0x2d2d2d2e ; ".-----
0x08048bd5 e836fcffff call sym.imp.puts ; int puts(const char *s)
0x08048bda c70424278e04. mov dword [esp], str.NEW_ACCOUNT_DETECTED ; [0x08048e27;4]=0x2121207c ; "| !! NEW ACCOUNT DETECTED !!"
0x08048be1 e82afcffff call sym.imp.puts ; int puts(const char *s)
0x08048be6 c70424cd8d04. mov dword [esp], str. ; [0x08048dcd;4]=0x2d2d2d7c ; "|-----|"
0x08048bed e81efcffff call sym.imp.puts ; int puts(const char *s)
0x08048bf2 c70424458e04. mov dword [esp], str.Input_your_serial ; [0x08048e45;4]=0x202d7e7c ; "|~- Input your serial: ~-|"
0x08048bf9 e812fcffff call sym.imp.puts ; int puts(const char *s)
0x08048bfe c70424098e04. mov dword [esp], str. ; [0x08048e09;4]=0x2d2d2d27 ; "'-----'"
0x08048c05 e806fcffff call sym.imp.puts ; int puts(const char *s)
0x08048c0a 8d442418 lea eax, [local_18h] ; 0x18 ; 24
0x08048c0e 89442404 mov dword [local_4h], eax
0x08048c12 c70424008d04. mov dword [esp], 0x08048d00 ; [0x08048d00;4]=0xa007525
0x08048c19 e842fcffff call sym.imp._isoc99_scanf ; int scanf(const char *format)

```

```
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x08048d00 2575 000a 0000 0000 1b5b 3332 6d2e 2d2d %u.....[32m.--
```

```

% 2) that it's not a multiple of a known prime. We don't
% check that rnd-1 is also coprime to all the known
% primes because there aren't many small primes where
% that's true. %/
for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
    if ((mods[i] + delta) % primes[i] == 0) {
        delta += 2;
        if (delta > maxdelta) {
            goto again;
        }
        goto loop;
    }
}
} else {
    for (i = 1; i < NUMPRIMES; i++) {
        %/ check that rnd is not a prime and also
        % that god(rnd-1,primes) = 1 (except for 2) %/
        if (((mods[i] + delta) % primes[i] == 0) ||

```

MBE Lab1A - Calling the authentication routine

```
static int probable_prime(BIGNUM *rnd, int bits) {  
    int i;  
    int t; mods[NUMPRIMES];  
    unsigned long delta;  
    unsigned long maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];  
    clear is_single_word = bits <= BN_BITS2;  
  
again:  
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {  
        return 0;  
    }  
}
```

```
0x08048c1e 8b442418 mov eax, dword [local_18h] ; [0x18:4]=-1 ; 24  
0x08048c22 89442404 mov dword [local_4h], eax  
0x08048c26 8d44241c lea eax, [local_1ch] ; 0x1c ; 28  
0x08048c2a 890424 mov dword [esp], eax  
0x08048c2d e8dddfffff call sym.auth  
0x08048c32 85c0 test eax, eax  
,=< 0x08048c34 751f jne 0x8048c55  
| 0x08048c36 c70424638e04, mov dword [esp], str.Authenticated ; [0x8048e63:4]=0x68747541 ; "Authenticated!"  
| 0x08048c3d e8cefbffff call sym.imp.puts ; int puts(const char *s)  
| 0x08048c42 c70424728e04, mov dword [esp], str.bin_sh ; [0x8048e72:4]=0x6e69622f ; "/bin/sh"  
| 0x08048c49 e8d2fbffff call sym.imp.system ; int system(const char *string)  
| 0x08048c4e b800000000 mov eax, 0  
,==< 0x08048c53 eb05 jmp 0x8048c5a  
|`-> 0x08048c55 b801000000 mov eax, 1  
| ; CODE XREF from main (0x8048c53)  
`-> 0x08048c5a 8b54243c mov edx, dword [local_3ch] ; [0x3c:4]=-1 ; '<' ; 60  
0x08048c5e 653315140000, xor edx, dword gs:[0x14]  
,=< 0x08048c65 7405 je 0x8048c6c  
| 0x08048c67 e894fbffff call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)  
`-> 0x08048c6c c9 leave  
0x08048c6d c3 ret
```

```
if (delta > maxdelta) {  
    goto again;  
}  
goto loop;  
}  
}  
} else {  
    for (i = 1; i < NUMPRIMES; i++) {  
        /* check that rnd is not a prime and also  
         * that gcd(rnd-1, primes) = 1 (except for 2) */  
        if (((mods[i] * delta) % primes[i]) != 1) {  
            delta += 2;  
            goto again;  
        }  
    }  
}
```

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    unsigned k;
    BIGNUM *candidate = BN_GENPRIME;
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
```

```

    }
} else {
    for (i = 1; i < NUMPRIMES; i++) {
        /* check that rnd is not a prime and also
         * that gcd(rnd-1,primes[i]) = 1 (except for 2) */
        if (((mod(rnd-1,primes[i]) != 1) || gcd(rnd-1,primes[i]) != 1) ||

```

MBE Lab1A - auth() 2/6: Antidebugging with ptrace

```

0x08048a5f  c744240c0000.  mov dword [local_ch_2], 0
0x08048a67  c74424080100.  mov dword [local_8h], 1
0x08048a6f  c74424040000.  mov dword [local_4h], 0
0x08048a77  c70424000000.  mov dword [esp], 0
0x08048a7e  e8edfdffff    call sym.imp.ptrace
0x08048a83  83f8ff        cmp eax, 0xffffffffffffffff
0x08048a86  752e          jne 0x8048ab6
0x08048a88  c70424088d04.  mov dword [esp], str.e_32m.-----+
0x08048a8f  e87cfdffff    call sym.imp.puts                ; int puts(const char *s)
0x08048a94  c704242c8d04.  mov dword [esp], str.e_31m_____TAMPERING_DETECTED ; [0x
0x08048a9b  e870fdffff    call sym.imp.puts                ; int puts(const char *s)
0x08048aa0  c70424508d04.  mov dword [esp], str.e_32m      ; [0x8048d50:4]=0x32335b1b
0x08048aa7  e864fdffff    call sym.imp.puts                ; int puts(const char *s)
0x08048aac  b801000000    mov eax, 1
0x08048ab1  e98c000000    jmp 0x8048b42

```

```

static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    BN_CTX *ctx = BN_CTX_new();
    BN_GENCB cb = BN_GENCB_create(0, 0, 0);
    BN_GENCB_set(&cb, 0, 0);
    BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD);
    /* we now have a random number 'rnd' to test. */
    delta = 2;
    if (delta > maxdelta) {
        goto again;
    }
    goto loop;
}
} else {
    for (i = 1; i < NUMPRIMES; i++) {
        /* check that rnd is not a prime and also
         * that gcd(rnd-1, primes) = 1 (except for 2) */
        if (((mod[1] * delta % prime[i]) < 1) ||
            ((mod[1] * delta % prime[i]) > 1)) {
            delta *= 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
}

```

MBE Lab1A - auth() 3/6: Pre-loop math

0x08048ab6
0x08048ab9
0x08048abc
0x08048abf
0x08048ac2
0x08048ac7
0x08048acc

8b4508
83c003
0fb600
0fbec0
3537130000
05eded5e00
8945f0

mov eax, dword [arg_8h]
add eax, 3
movzx eax, byte [eax]
movsx eax, al
xor eax, 0x1337
add eax, 0x5e00
mov dword [local_10h], eax

```
static int probable_prime(BIGNUM *rnd, int bits) {  
    int i;  
    uint64_t mods[NUMPRIMES];  
    BN_ULONG delta;  
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];  
    char is_single_word = bits <= BN_BITS2;  
  
again:  
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {  
        return 0;  
    }  
  
    /* we now have a random number 'rnd' to test. */  
    for (i = 1; i < NUMPRIMES; i++) {  
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);  
        if (mod == (BN_ULONG)-1) {  
            return 0;  
        }  
  
        /* 2) That it's not a multiple of a known prime. We don't  
        * check that rnd-1 is also coprime to all the known  
        * primes because there aren't many small primes where  
        * that's true. */  
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {  
            if (((mod[i] + delta) % primes[i]) == 0) {  
                delta += 2;  
                if (delta > maxdelta) {  
                    goto again;  
                }  
                goto loop;  
            }  
        }  
    } else {  
        for (i = 1; i < NUMPRIMES; i++) {  
            /* check that rnd is not a prime and also  
            * that gcd(rnd-1, primes) = 1 (except for 2) */  
            if (((mod[i] + delta) % primes[i]) != 1) {  
                delta += 2;  
                if (delta > maxdelta) {  
                    goto again;  
                }  
                goto loop;  
            }  
        }  
    }  
}
```

MBE Lab1A - auth() 4/6: Loop header, restricting chars

	0x08048acf	c745ec000000	mov dword [local_14h], 0
,=<	0x08048ad6	eb4e	jmp 0x8048b26
.,---->	0x08048ad8	8b55ec	mov edx, dword [local_14h]
:	0x08048adb	8b4508	mov eax, dword [arg_8h]
:	0x08048ade	01d0	add eax, edx
:	0x08048ae0	0fb600	movzx eax, byte [eax]
:	0x08048ae3	3c1f	cmp al, 0x1f
,=====<	0x08048ae5	7f07	jg 0x8048aee
l:	0x08048ae7	b801000000	mov eax, 1
,=====<	0x08048aec	eb54	jmp 0x8048b42

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    if (bits < 16) {
        mode = NUMPRIMES;
        maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
        if (is_single_word = bits <= BN_BITS2;
            again;
            if (BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
                return 0;
            }

            /* we now have a random number 'rnd' to test. */
            for (i = 1; i < NUMPRIMES; i++) {
                BN_ULONG mod = BN_rand_word(rnd);
                if ((BN_ULONG)primes[i] < mod) {
                    goto again;
                }
                if ((mod % primes[i] == 0) && primes[i] < rnd_word) {
                    if ((mod[i] + delta) % primes[i] == 0) {
                        delta += 2;
                        if (delta > maxdelta) {
                            goto again;
                        }
                        goto loop;
                    }
                }
            }
        } else {
            for (i = 1; i < NUMPRIMES; i++) {
                /* check that rnd is not a prime and also
                 * that god(rnd-1) == 1 (except for 2) */
                if (((mod[i] + delta) % primes[i] == 0) && primes[i] < rnd_word) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                    goto loop;
                }
            }
        }
    }
}
```

MBE Lab1A - auth() 5/6: Loop body, much math

`----->	0x08048aee	8b55ec	mov edx, dword [local_14h]
:	0x08048af1	8b4508	mov eax, dword [arg_8h]
:	0x08048af4	01d0	add eax, edx
:	0x08048af6	0fb600	movzx eax, byte [eax]
:	0x08048af9	0fbec0	movsx eax, al
:	0x08048afc	3345f0	xor eax, dword [local_10h]
:	0x08048aff	89c1	mov ecx, eax
:	0x08048b01	ba2b3b2388	mov edx, 0x88233b2b
:	0x08048b06	89c8	mov eax, ecx
:	0x08048b08	f7e2	mul edx
:	0x08048b0a	89c8	mov eax, ecx
:	0x08048b0c	29d0	sub eax, edx
:	0x08048b0e	d1e8	shr eax, 1
:	0x08048b10	01d0	add eax, edx
:	0x08048b12	c1e80a	shr eax, 0xa
:	0x08048b15	69c039050000	imul eax, eax, 0x539
:	0x08048b1b	29c1	sub ecx, eax

MBE Lab1A - auth() 6/6: Loop footer, return targets

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    int32_t mod[NUMPRIMES];
    BN_CTX *ctx = BN_CTX_new();
    BN_mask2b(rnd, BN_MASK2 - primes[NUMPRIMES - 1]);
    char is_single_word = bits <= BN_BITS2;

again:
    if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
```

```

| :||| 0x08048b22 8345ec01 add dword [local_14h], 1
| :||| ; CODE XREF from sym.auth (0x8048ad6)
| :||`-> 0x08048b26 8b45ec mov eax, dword [local_14h]
| :|| 0x08048b29 3b45f4 cmp eax, dword [local_ch]
| `====< 0x08048b2c 7caa jl 0x8048ad8
| || 0x08048b2e 8b450c mov eax, dword [arg_ch]
| || 0x08048b31 3b45f0 cmp eax, dword [local_10h]
| ||,=< 0x08048b34 7407 je 0x8048b3d
| ||| 0x08048b36 b801000000 mov eax, 1
| ,====< 0x08048b3b eb05 jmp 0x8048b42
| |||`-> 0x08048b3d b800000000 mov eax, 0
| ||| ; CODE XREFS from sym.auth (0x8048a5a, 0x8048ab1, 0x8048a
| -```--> 0x08048b42 c9 leave
| 0x08048b43 c3 ret

```

```

    }
} else {
    for (i = 1; i < NUMPRIMES; i++) {
        /* check that rml is not a prime and also
         * that gcd(rml-1, primes[i]) = 1 (except for 2) */
        if (((mod(rml-1, primes[i]) != 1) ||
            (primes[i] == 2 && mod(rml-1, 2) == 0)) {

```

MBE Lab1A - Z3ing auth() 1/?: Setting up variables

```
import z3
solver = z3.Solver()
wanted_length = 6
assert wanted_length > 5 # checked at 0x08048a4f
sym_username = [z3.BitVec('x{i}'.format(i=i), 8) for i in range(wanted_length)]
sym_serial = z3.BitVec('serial', 32)
```

- ▶ 8-bit entries for each character
- ▶ 32-bit serial number
- ▶ Concrete input length: z3.Array exists, but is more expensive
- ▶ Only use z3.Array if you need symbolic indexing

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG mod;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    clear_is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) < 1)
        return 0;
}

/* we now have a random number 'rnd' to test. */
for (i = 1; i < NUMPRIMES; i++) {
    BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
    if (mod == (BN_ULONG)-1) {
        return 0;
    }
    mods[i] = (uint16_t)mod;
}
/* If bits is so small that it fits into a single word then we
 * additionally don't want to exceed that many bits. */
if (is_single_word) {
    BN_ULONG size_limit;
    if (bits == BN_BITS2) {
        size_limit = ((BN_ULONG)0) - get_word(rnd);
    } else {
        size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
        maxdelta = size_limit;
    }
}
delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    a prime number because it's a multiple of
         * 2) that it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

MBE Lab1A - Z3ing auth() 2/?: Translating the pre-loop math

```
eax = sym_username[3]
eax ^= z3.BitVecVal(0x1337, 32)
eax += z3.BitVecVal(0x5eeded, 32)
local_10h = eax
```

- ▶ We're wrapping concrete values in `z3.BitVecVal` so that wrapping/truncation happens the x86 way
- ▶ If we were using python longs here, we'd have to manually mask them back into range

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG maxdelta = 0;
    BN_ULONG mindelta = 1;
    char is_single_word = bits <= BN_BITS2;

again:
    if (BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
        delta = 0;
    }

loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that it's possible prime is a single word then
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

Resources

- ▶ <https://github.com/Z3Prover/z3/>
- ▶ <https://pypi.org/project/z3-solver/>
- ▶ <https://rise4fun.com/Z3/tutorialcontent/guide>
- ▶ https://en.wikipedia.org/wiki/Satisfiability_modulo_theories
- ▶ <https://github.com/RPISEC/MBE>

```
static int static_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) &
        return 0;
    }

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)-1) {
            return 0;
        }
        mods[i] = (uint16_t)mod;
    }
    /* If bits is so small that it fits into a single word then we
     * additionally don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            delta = size_limit;
        }
    } else {
        delta = 0;
    }
loop:
    if (is_single_word) {
        BN_ULONG rnd_word = get_word(rnd);

        /* In the case that the candidate prime is a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3 as being a prime number because it's a multiple of
         *    three.
         * 2) That it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```